SOFTWARE REFERENCE MANUAL

PCOMM32PRO

Communication Driver Set

3A0-09WPRO-xSx2

January 28, 2003

DELTA TAU
Data Systems, Inc.
NEW IDEAS IN MOTION ...

Single Source Machine Control *Power // Flexibility // Ease of Use*
21314 Lassen Street Chatsworth, CA 91311 // Tel. (818) 998-2095 Fax. (818) 998-7807 // www.deltatau.com

\

Table of Contents

INTRODUCTION TO USER MANUAL....1

ABOUT PCOMM32PRO....1
A GLOBAL VIEW OF THE LIBRARY....1
SUPPORTED OPERATING SYSTEMS....2
COMMUNICATION MODES....2
Plug & play ports....2
Non-plug & play ports....2

GETTING STARTED....4

SETTING UP COMMUNICATIONS WITH PMAC....4
INSTALLING PCOMM32PRO....4
USAGE OF PCOMM32PRO....4
Linking Your Code with the Driver....4
Initializing Communication....5
PmacServer....7
Shutting Down Communication....7
A Guide to Using ASCII Communication Functions....7
Common Problems Experienced Using ASCII Communications Functions....7
Thread-Safe ASCII Communications....8
Error Handling - ASCII Communication And Other Functions....8
Int vs long Data Type....10

PCOMM32PRO FEATURES....11

USING INTERRUPTS....11
Send A Message To A Window....11
Call A Function....12
Start A Thread Function....12
Set An Event....12
DOWNLOADING TO PMAC....12

PCOMM32PRO DPR FEATURES....14

A GLOBAL VIEW OF THE DPR SUPPORT FUNCTIONS....14
Control Panel (Non-Turbo PMACs)....14
Fixed Real Time Data buffer....14
Fixed Background Data Buffer....15
Variable Background Data Buffer....15
Binary Rotary Buffer....15
Read/Write Functions....15
INITIALIZATION /SHUTDOWN....15
MEMORY ORGANIZATION/CORRECT ORDER OF INITIALIZATION....15
USING THE DPR CONTROL PANEL (NON-TURBO PMAC)....17
Action Bits....17
Request Bits....17
Assigning A New Feedrate Override....18
Feedrate Override Enable Bits....19
USING THE DPR REAL TIME FIXED DATA BUFFER....19
USING THE DPR REAL TIME FIXED DATA BUFFER (TURBO)....21
USING THE DPR BACKGROUND FIXED DATA BUFFER....24
USING THE DPR BACKGROUND FIXED DATA BUFFER (TURBO)....26
USING THE DPR VARIABLE BACKGROUND READ/WRITE DATA BUFFER....29
Variable Read Data Buffer....29
Variable Write Data Buffer....32
USING THE DPR BINARY ROTARY MOTION PROGRAM BUFFER....32

PmacDPRRotBufChange() 33
PmacDPRRotBufRemove() 33
PmacDPRRotBufClr() 33
PmacDPRRotBuf() 33
PmacDPRBufLast() 33
Downloading "Stuffing" functions 34

INTRODUCTION TO PROGRAMMER'S REFERENCE 38

IMPORTANT INFORMATION ABOUT METHOD OF COMMUNICATION BEING USED BY PCOMM32PRO 38
MAXIMUM NUMBER OF RESPONSE BUFFER CHARACTERS 38

INITIALIZATION, SHUTDOWN AND CONFIGURATION FUNCTIONS 39

OPENPMACDEVICE() 39
CLOSEPMACDEVICE() 39
PMACSELECT() 39
LOCKPMAC() 40
RELEASEPMAC() 41
PMACSETCRITICAIVARS() 41
PMACABORTDPRTEST() 41
PMACCONFIGURE() 42
PMACGETDPRAMAVAILABLE() 42
PMACINBOOTSTRAPMODE() 42
PMACDPRTEST() 43
PMACSERGETPORT() 43

ASCII COMMUNICATION FUNCTIONS 44

GETUSRESPONSE(), USREADREADY() 44
PMACFLUSH() 44
PMACGETASCIICOMM() 45
PMACGETBUFFERA() 46
PMACGETCONTROLRESPONSEA(), PMACGETCONTROLRESPONSEEXA() 46
PMACGETLINEA(), PMACGETLINEEXA() 46
PMACGETRESPONSEA(),PMACGETRESPONSEEXA() 47
PMACREADREADY() 48
PMACSENDCHARA() 48
PMACSENDCOMMANDA() 48
PMACSENDLINEA() 49
PMACSETASCIICOMM() 50

DOWNLOADING FUNCTIONS 51

PMACABORTDOWNLOAD() 51
PMACADDDOWNLOADFILEA() 51
PMACDOWNLOADA() 52
PMACDOWNLOADFIRMWAREFILE() 53
PMACMULTIDOWNLOAD() 53
PMACREMOVEDOWNLOADFILE() 54
PMACSETMAXDOWNLOADERRORS() 55

DUAL PORTED RAM CONTROL PANEL 56

PMACDPRCONTROLPANEL() 56
PMACDPRGETACTIONBIT() AND PMACDPRSETACTIONBIT() 56
PMACDPRGETFOENABLEBIT() AND PMACDPRSETFOENABLEBIT() 57
PMACDPRGETFOVALUE() AND PMACDPRSETFOVALUE() 57
PMACDPRGETREQUESTBIT() AND PMACDPRSETREQUESTBIT() 58

DPR REAL TIME FIXED DATA BUFFER 60

DPR Real Time Fixed Data Buffer Initialization and Handshaking 60
DPR Real Time Fixed Data Buffer Query Routines 60

DPR REAL TIME FIXED DATA BUFFER (TURBO) 64

DPR Real Time Fixed Data Buffer Initialization (Turbo) 64
DPR Real Time Fixed Data Buffer Query Routines (Turbo) 64

DPR BACKGROUND FIXED DATA BUFFER 68

DPR Background Fixed Data Buffer Initialization and Handshaking 68
DPR Background Fixed Data Buffer Query Routines 69

DPR BACKGROUND FIXED DATA BUFFER (TURBO) 72

DPR Background Fixed Data Buffer Initialization and Handshaking (Turbo) 72
DPR Background Fixed Data Buffer Query Routines (Turbo) 72

DPR VARIABLE BACKGROUND READ/WRITE DATA BUFFER 74

PmacDPRBackground() 74
PmacDPRBufLast() 74
PmacDPRGetVBGAddress() 75
PmacDPRGetVBGNumEntries() 75
PmacDPRGetVBGDataOffset() 75
PmacDPRGetVBGAddrOffset() 76
PmacDPRGetVBGServoTimer() 76
PmacDPRGetVBGStartAddr() 76
PmacDPRGetVBGTotalEntries() 77
PmacDPRVarBufChange() 77
PmacDPRVarBufInit() 77
PmacDPRVarBufRead() 78
PmacDPRVarBufReadEx() 78
PmacDPRVarBufRemove() 79
PmacDPRWriteBufferEx() 79

DPR BINARY ROTARY BUFFER FUNCTIONS 81

PmacDPRAsciiFileToRot() 81
PmacDPRAsciiStrToRot() 82
PmacDPRAsciiStrToRotEx() 82
PmacDPRRotBufChange() 83
PmacDPRRotBufRemove() 84
PmacDPRRotBufClr() 84
PmacDPRRotBuf() 85

DPR NUMERIC READ AND WRITE 86

General Information 86
Standard Read/Write 86
Masking 86
Dual Word Conversion 86
PmacDPRDWordBitSet() 87
PmacDPRFloat() 88
PmacDPRGetDWord() 88
PmacDPRGetDWordMask() 88
PmacDPRGetFloat() 88
PmacDPRGetMem() 89
PmacDPRGetWord() 89
PmacDPRResetDWordBit() 90
PmacDPRSetDWord() 90
PmacDPRSetDWordBit() 90

PMACDPRSETDWORDMASK()....90
PMACDPRSETFLOAT()....91
PMACDPRSETMEM()....91
PMACDPRSETWORD()....92

DATA GATHERING FUNCTIONS....93

PMACSETGATHER....93
PMACSETQUICKGATHEREX....93
PMACGETGATHER....93
PMACSTARTGATHER....94
PMACSTOPGATHER....94
PMACSETQUICKGATHERWITHDIRECTCURRENTEX....95

INTERRUPT FUNCTIONS....97

GENERAL INFORMATION INTERRUPT FUNCTIONS....97
PMACINTRFIREEVENTINIT()....97
PMACINTRFUNCCALLINIT()....97
PMACINTRRUNTHREADINIT()....98
PMACINTRTERMINATE()....99
PMACINTRWNDMSGINIT()....99

SERIAL CHECKSUM COMMUNICATION FUNCTIONS....100

PMACSERDOCHECKSUMS()....100
PMACSERCHECKRESPONSEA()....100
PMACSERCHECKSENDLINEA()....100
PMACSERCHECKGETLINEA()....101
SOURCE/EXAMPLE PMACSERCHECKRESPONSE()....101

VARIABLE FUNCTIONS....103

PMACGETVARIABLE()....103
PMACSETVARIABLE()....103

UTILITY FUNCTIONS....105

PMACGETROMDATEA()....105
PMACGETROMVERSIONA()....105
PMACGETUSERHANDLE()....105
PMACGETPMACTYPE()....106
INP()....106
OUTP()....106
GETBITVALUE()....107
HEX_LONG2()....107

DATA TYPES, STRUCTURES, CALLBACKS, AND CONSTANTS....108

DOWNLOADGETPROC....108
DOWNLOADPROGRESS....108
DOWNLOADMSGPROC....109
DPRTESTMSGPROC....109
DPRTESTPROGRESS....109
DPR_BIN_ROT_ERRORS....110
GLOBALSTATUS....111
MOTION....112
MOTIONMODE....112
PROGRAM....112
SERVOSTATUS....112

EXTENDED FUNCTIONS....113

PMACDPRREALTIMEEX....113
PMACDPRBACKGROUNDEX()....114
PMACDPRVARBUFINITEX()....115
PMACDPRVARBUFCHANGEEX()....115
PMACDPRWRITEBUFFEREX()....116

COMMUNICATION APPLICATION NOTES....119

COMMON COMMUNICATION TRAPS....119
COMMON DUAL PORTED RAM TRAPS....122
TERMINAL EXAMPLE PROGRAM....124

INDEX....129

USER MANUAL

INTRODUCTION TO USER MANUAL

About PComm32PRO

The Delta Tau PComm32PRO Communication Driver is a set of more than 400 functions written as a development tool for the creation of PMAC applications on Windows 98/ME/2000 and XP. The routines were designed with robustness, speed and portability in mind. PComm32PRO may be used by all PMAC types.
Methods of communication include the bus (ISA and PCI), Dual Ported RAM, Serial, USB and Ethernet.

PComm32PRO is compatible with the 32 bit Borland and Microsoft development products, which include:

- Microsoft Visual C/C++
- Microsoft Visual Basic
- Borland Delphi
- Borland C++ Builder

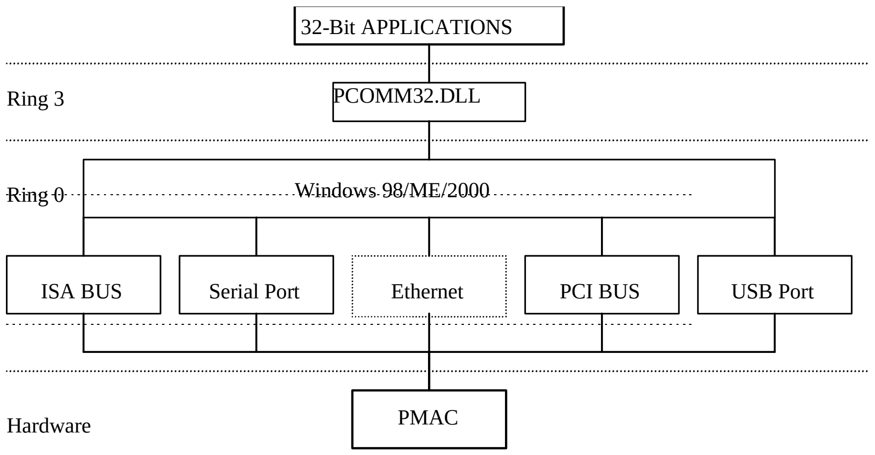
This manual assumes that you know Windows basics and general programming practices.

A Global View of the Library

PComm32PRO can be used for Windows 98/ME/2000 and XP application development. Use of this driver is the same regardless of what operating system is in use.
PComm32PRO itself consists of three sets of files.

- PCOMM32.DLL **-** A 32-bit DLL.
- PMAC**x**.SYS (where **x = ISA**, **SER**, **PCI**, **ETH**, or **USB, USBL and USBT**) **-** Windows 98/ME or Windows 2000/XP kernel drivers.
- PMAC**x**.INF (where **x = ISA**, **SER**, **PCI**, or **USB**) - Windows Setup Information files.
- PMACSERVER.EXE - Running in the background when any application is communicating to the PMAC.
- USBCONFIGURE.EXE and ETHCONFIGURE.EXE **-** responsible for configuration of the USB and Ethernet communication boards (including loading the appropriate firmware and assignment of an IP address [Ethernet] or Serial number [USB] along with other settings such as ACC54E, PC104, QMAC or CPCI baords and creating other related registry values.) For Ethernet mode of communication, the choice between TCP and UDP modes of communication is also provided by ETHCONFIGURE.EXE.

The illustration below shows how these modules are related.



PComm32PRO Driver Structure

Supported operating systems

- Windows 98
- Windows 2000
- Windows ME
- Windows XP

Communication Modes

Plug & play ports

- PCI BUS PMAC
- USB Port PMAC

Non-plug & play ports

- ISA Bus PMAC
- Serial Port PMAC
- Ethernet port PMAC

Using Pcomm32PRO

GETTING STARTED

Setting up Communications with PMAC

No applications, including all Delta Tau software programs, will be used to add PMACs in your system. Rather, communication settings have been centralized in your operating system, making the set up of each PMAC much like that of other devices in your computer (i.e. printer, video card, sound card, etc.) All setup is done through the Control Panel's **Add New Hardware** Wizard. The wizard will help install and register the just-inserted devices. It is important that all applications that use PComm32PRO be shut down. This includes Pewin32PRO, NC for Windows, and any applications developed with PComm32PRO or PTalkDTPRO (ActiveX component for PMAC).

Installing Pcomm32PRO

Before installing Pcomm32PRO, read the license agreement included in this manual (behind title page), and make a backup copy of the installation disks. Then install Pcomm32PRO according to the instructions in the User Manual ***"Pcomm32PRO Installation Procedures."***

Usage of PComm32PRO

In this section we discuss the usage of PComm32PRO in general, as well as specifically through an example program written with Microsoft Visual C++ (MSVC).
The example MSVC program, called PMACTEST, uses the Microsoft Foundation Class (MFC) library. The important details of linking with the library, as well as initializing and shutting down communication are discussed in this section. The PMACTEST project is located in the "<Install Directory>\PmacTest\" directory.

Linking Your Code with the Driver

The driver interface is done through a Dynamic Link Library (PCOMM32DLL), which is a standard windows library file format. Your application must interface with the DLL to communicate to PMAC. Methods for interfacing to DLLs for any given development environment are well documented; so those details won't be discussed here. At a minimum, the function prototypes and the location of the DLL in the operating system are needed to interface to a DLL. The prototypes for the PCOMM32.DLL functions are detailed in the reference manual, as well as the header files that come with the library. The installation program of the driver places the PComm32.dll in the Windows system directory.
The PMACTEST program uses the LoadLibrary() and GetProcAddress() Win32 system API calls to link with the PComm32.dll. In the RUNTIME.CPP source file you will find a routine, RuntimeLink() which performs these tasks. A small excerpt from the file is show below:

```
#include "runtime.h"

HINSTANCE RuntimeLink()

{

  // Get handle to PCOMM32.DLL

  hPmacLib = LoadLibrary(DRIVERNAME);

//  Get some DLL procedure addresses,  for example OpenPmacDevice and ClosePmacDevice
                                DeviceOpen =
                                (OPENPMACDEVICE)GetProcAddress(hPmacLib,“OpenPmacDevice");

                                DeviceClose =
                                (CLOSEPMACDEVICE)GetProcAddress(hPmacLib,“ClosePmacDevice");

  .

  .
// Check validity of the procedure addresses
  if(!DeviceOpen || !DeviceClose ||){
        // Report an error, couldn’t link with one or more functions
        return NULL
  }
  else
    return hPmacLib;
}
```

The header file, “runtime.h” has the type definitions for the function pointers used in the code above. A portion of the file is shown below:

```
#define DRIVERNAME TEXT("PComm32.dll") //name of the user-dll driver


//*******************************************************************
// COMM Type Defines
//*******************************************************************
typedef BOOL (CALLBACK* OPENPMACDEVICE)(DWORD dwDevice);
typedef BOOL (CALLBACK* CLOSEPMACDEVICE)(DWORD dwDevice);
.
.
.
```

Note:

It is wise to link only with those functions that your program uses.

Initializing Communication

In the demo program PMACTEST a class has been created, CPmacTestDoc (see pmactestdoc.cpp), which has been designed to handle all communication between the application and PMAC. It establishes communication by a member function called InitDocument(). InitDocument() is shown below:

```
BOOL CPmacTestDoc::InitDocument(LPCTSTR lpszPathName)
{
  TCHAR str[_MAX_PATH], vs[30],ds[30];

  // Get handle to PCOMM32.DLL, and get procedure addresses
  if(!m_hPmacLib)
     m_hPmacLib = RuntimeLink();

  if(m_hPmacLib == NULL)
     return FALSE;// We failed to load the library or functions

  if(m_bDriverOpen)// Only call OpenPmacDevice() once
     return TRUE;
// Call OpenPmacDevice()
  m_bDriverOpen = DeviceOpen(m_dwDevice);

  if(m_bDriverOpen) {
     DeviceGetRomVersion(m_dwDevice,vs,30);
     DeviceGetRomDate(m_dwDevice,ds,30);
     sprintf(str,TEXT("V%s %s"),vs,ds);
     SetTitle(str);
     return TRUE;
  }
  else {
     AfxMessageBox("Could not initialize PMAC Comm.");
     SetTitle(TEXT("Not Linked.."));
     return FALSE;
  }
}
```

Notice the two steps to initializing communication to a PMAC:

1. Load the DLL, and get procedure addresses.
2. Call OpenPmacDevice() with the PMAC device number.

The PMAC device number is 0 for the first PMAC in your system, 1 for the second and so on. Pmac device configuration within the operating system is done with dialogs created via the PmacSelect() function.

Once PMAC communication is initialized, all other functions in the library may be called.

PmacServer

The PComm32PRO driver uses an application, PmacServer, to handle continuous global memory updates of the DPR automatic functions. PmacServer is launched whenever the driver opens communication to any PMAC device. PmacServer will eventually shut itself down after all communications have been shut down. You may want to disable the continuous global memory updates by using the properties button in the PmacSelect() dialog box. This change should not affect your development environment.

Shutting Down Communication

To release any resources the driver has allocated, it is good programming practice to shut down any communication links that have been opened. In PMACTEST, the demo application, this is done within the CPmacTestDoc class as shown below:

```
void CPmacTestDoc::CloseDocument()
{
  if(m_bDriverOpen) {
     // Call ClosePmacDevice()
      m_bDriverOpen = !DeviceClose(m_dwDevice);
  }
  if(m_hPmacLib) {// Free the library
    FreeLibrary(m_hPmacLib);
    m_hPmacLib = NULL;
  }
}
```

Notice the two steps in shutting down communication:

1. Shut down the PComm32PRO driver with a call to ClosePmacDevice()
2. Call FreeLibrary so that the operating system may unload the PCOMM32.DLL from memory

A Guide to Using ASCII Communication Functions

Most if not all of your communication with the PMAC can be handled with the PmacGetResponseA() function. This function will send a command string (i.e. “#1j+, “?”, “Open Prog1”, etc) to the PMAC and retrieve and place any pending responses within a response buffer for your use. This is an efficient and *safe* function to use.   The word *safe* is emphasized because there are functions (i.e. PmacSendCharA(), PmacGetLineA() and PmacSendLineA()) which, if not used properly can cause un-synchronized communication (the mismatching of PMAC Commands to PMAC responses).   PmacGetResponseA() always matches the command string with the response string or else it “times out.”
For getting responses to a PMAC control-character command it’s easiest to use PmacGetControlResponseA().

Common Problems Experienced Using ASCII Communications Functions

This section outlines some of the more frequently encountered issues with solutions. Also see “Communication Application Notes.”

Modifying Critical PMAC I-Variables

There are several I-Variables that PComm32Pro expects or enforces to certain values. The table below describes each and their purpose. Do not modify these I-Variables.

I Variable	Meaning	Desired Value
I3	Handshaking mode	2
I4	Checksummed Serial Communication Enable/Disable	0 or 1
I6	Error Reporting Mode	1
I63	Control-X echo	1
I64	Unsolicited Response Tagged	1

Thread-Safe ASCII Communications

PComm32PRO is a thread-safe communication driver if used as described in this section. When using multiple threads, there is the possibility of two or more threads trying to communicate with the same PMAC at the same time. Two routines within the driver, LockPmac() and ReleasePmac() are used to mutually exclude other threads from continuing to run code that may be problematic. LockPmac() and ReleasePmac() are used internally with functions such as PmacGetResponseA() and DownloadFile(). This means that two or more threads, even two or more applications, may be communicating to the same PMAC through the same method (bus, USB etc.) and not have a synchronization problem.

The LockPmac() routine asks the operating system if the specified PMAC resource is available. If it is, the operating system will now restrict usage to the PMAC to that calling thread. If the PMAC resource is not available, the operating system will put the calling thread in a wait state until it does become available.

The ReleasePmac() routine releases the PMAC resource for other threads to access. It is critical to always release for every lock call.

You may find that you need to communicate to PMAC, uninterrupted by other threads or applications, from time to time. Use LockPmac() and ReleasePmac() routines for these instances.

If your application suddenly locks up, try to recall when and where you call LockPmac() and ReleasePmac(). Probably you did not release as many times as you locked.

Error Handling - ASCII Communication And Other Functions

Extended error handling is implemented within the ASCII communication routines that have the **Ex** suffix:

```
long PmacGetLineExA(DWORD dwDevice, PCHAR response, UINT maxchar);
long PmacGetResponseExA(DWORD dwDevice,PCHAR response,UINT maxchar,PCHAR command);
long PmacGetControlResponseExA(DWORD dwDevice,PCHAR response,UINT maxchar, CHAR ctl_char);
```

Similarly named routines shown below do not have extended error handling:

```
long PmacGetLineA(DWORD dwDevice, PCHAR response, UINT maxchar);
long PmacGetResponseA(DWORD dwDevice,PCHAR response,UINT maxchar,PCHAR command);
long PmacGetControlResponseA(DWORD dwDevice,PCHAR response,UINT maxchar, CHAR ctl_char);
```

The only difference between the two sets of functions is the return value. The extended routines provide error status (in the most significant byte) in addition to the number of characters received (all other bytes), whereas the non-Ex routines simply return the number of characters received from PMAC.

It is strongly recommended that users use the extended functions to allow for enhanced error handling.

The following error status codes exist for the extended ASCII communication routines:

Below are all negative return codes

Mnemonic	Value	Meaning
COMM_EOT	0x80000000	An acknowledge character (ACK ASCII 9) was received indicating end

		of transmission from PMAC to Host PC.
COMM_TIMEOUT	0xC0000000	A timeout occurred. The time for the PC to wait for PMAC to respond had been exceeded.
COMM_BADCKSUM	0xD0000000	Used when using Checksum communication. If a bad checksum occurred this error will be returned.
COMM_ERROR	0xE0000000	Unable to communicate.
COMM_FAIL	0xF0000000	Serious failure.
COMM_ANYERR	0x70000000	Some error occurred.
COMM_UNSOLICITED	0x10000000	An unsolicited response has been received from PMAC. Usually caused by PLC's or Motion Programs that have "SEND" or "COMMAND" statements.

The mnemonics above, in addition to MACROs to parse the return value, are defined in the provided mioctl.h header file. To get at the individual portions of the return value the following MACROs are helpful:

```
#define COMM_CHARS(c)    (c & 0x0FFFFFFF) // Returns the number of characters
#define COMM_STATUS(c)   (c & 0xF0000000) // Returns the status byte
```

To check for individual error codes the MACROs below are very useful:

```
#define IS_COMM_MORE(c)     ((c & COMM_FAIL) == 0)
#define IS_COMM_EOT(c)      ((c & COMM_FAIL) == COMM_EOT)
#define IS_COMM_TIMEOUT(c)  ((c & COMM_FAIL) == COMM_TIMEOUT)
#define IS_COMM_BADCKSUM(c) ((c & COMM_FAIL) == COMM_BADCKSUM)
#define IS_COMM_ERROR(c)    ((c & COMM_FAIL) == COMM_ERROR)
#define IS_COMM_FAIL(c)     ((c & COMM_FAIL) == COMM_FAIL)
#define IS_COMM_ANYERROR(c) ((c & COMM_ANYERR) > 0)
#define IS_COMM_UNSOLICITED(c) ((c & 0xF0000000) == COMM_UNSOLICITED)
```

If you are using the non-extended routines you may use the PmacGetError() routine. After a function call the PmacGetError() will return 0 if all went well. Otherwise the non-zero value will represent the error code. A corresponding error string can be retrieved via the PmacGetErrorStrX() routine. The error string is not reset after every call to a communication function, but the Error number is.

Int vs long Data Type

Visual Basic: integer variables are stored as 16-bit (2-byte) numbers ranging in value from -32,768 to 32,767.
Visual C++: The size of a signed or unsigned **int** item is the standard size of an integer on a particular machine. For example, in 16-bit operating systems, the **int** type is usually 16 bits, or 2 bytes. In 32-bit operating systems, the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent to either the **short int** or the **long int** type, and the **unsigned int** type is equivalent to either the **unsigned short** or the **unsigned long** type, depending on the target environment. The **int** types all represent signed values unless specified otherwise.
C++Builder: On 32-bit platforms, the keyword **int** specifies a 32-bit signed integer. On 16-bit platforms, the keyword **int** is an optional keyword that can accompany the keywords **small**, **short**, and **long**.
PComm32PRO the 32-bit Delta Tau driver has been developed with Visual C++ for 32-bit operating systems; therefore, **int** has always been 4 bytes long. In order to make our code generic under all development environments, this document recommends that all **int** types be changed to **long** at least for the Visual Basic development environment.

PCOMM32PRO FEATURES

Using Interrupts

Interrupts are provided for all Windows 95/98 and Windows NT operating systems. There are four methods of interrupt notification for your program:

1. Send a message to a window.
2. Call a function.
3. Start a thread function.
4. Set an event.

Each method has a separate initialization function. Common to all initialization functions is the parameter *ulMask*. This parameter determines the interrupt service vector(s) to be used for the interrupt initiated by the function.

The least significant byte of *ulMask* controls which conditions will generate an interrupt. A bit value of 0 enables, 1 disables.

Bit	**PMAC Signal**
0	In Position of Coordinate System
1	Buffer Request (PMAC's request for more moves)
2	Error, A motor(s) in the coordinate system has had a fatal following error
3	Warning, A motor(s) in the coordinate system has had a warning following error
4	Host Request, PMAC has an ASCII response for the host
5-7	User programmable, see PMAC User's Guide, Writing a Host Communications Program

The PmacINTRTerminate() function is provided to shut down the interrupt service.

Send A Message To A Window

This interrupt method uses the Windows SendMessage() function to send a message to a window handle (HWND). Both the message and the windows handle are provided by the programmer. Use the function:

```
BOOL  CALLBACK PmacINTRWndMsgInit(DWORD dwDevice, HWND hWnd, UINT msg, ULONG ulMask);
```

to initiate this method.

Call A Function

This interrupt method causes a function provided by the programmer to be called. Use the function:

```
BOOL  CALLBACK PmacINTRFuncCallInit(DWORD dwDevice, PMACINTRPROC pFunc, DWORD msg, ULONG
       ulMask);
```

to initiate this method.

Start A Thread Function

This interrupt method starts a thread function provided by the programmer. It is the programmer's responsibility to terminate the thread. Use the function:

```
BOOL  CALLBACK PmacINTRRunThreadInit(DWORD dwDevice, LPTHREAD_START_ROUTINE pFunc, UINT msg,
       ULONG ulMask);
```

to initiate this method.

Set An Event

This interrupt method sets an event provided by the programmer. It is the programmer's responsibility to reset the event. Use the function:

```
BOOL  CALLBACK PmacINTRFireEventInit(DWORD dwDevice,HANDLE hEvent,ULONG ulMask);
```

to initiate this method.

Downloading To PMAC

Downloading ASCII PMAC Data

Downloading of PMAC motion, plc, configuration files etc. may be done by the using the PmacDownload() function. This function can:

- Parse Macros (i.e. #define, #include etc.)
- Compile PLC's to PLCC's
- Create a map file from macros
- Create a log file of download progress
- Invoke a callback function for displaying progress (same text as the log file that can be created)
- Download a file or a buffer through a line retrieval call back function
- Update a progress bar through a call back function

Multiple File Cooperative Downloading

A problem exists when downloading PLCC programs to the PMAC from multiple applications. All PLCC programs that should be running within PMAC should be downloaded at the same time (in other words, all in one file). When application "A" creates and downloads a file with PLCCs, it has no way of knowing that the PLCC-bearing files of other applications in the system must be downloaded as well. The solution is to have a central location for which all applications in the system register the set of files they use. From this registered set, any one of the cooperating applications can download all plcc programs required by all cooperating applications. Download procedures have been developed to utilize this solution and eliminate any problems associated with PMAC compiled PLCs and PLCCs.
To remain compatible with any products that Delta Tau and Third party vendors create, you must implement the following set of functions, instead of the standard Download() procedures:

`PmacAddDownloadFileA()`
`PmacMultiDownloadA()`
`PmacRemoveDownloadFileA()`
The PmacAddDownloadFileA() appends to a centrally located list of files to be downloaded every time PmacMultiDownloadA() is called to download files to PMAC.   PmacRemoveDownloadFileA() will remove the files from the list.

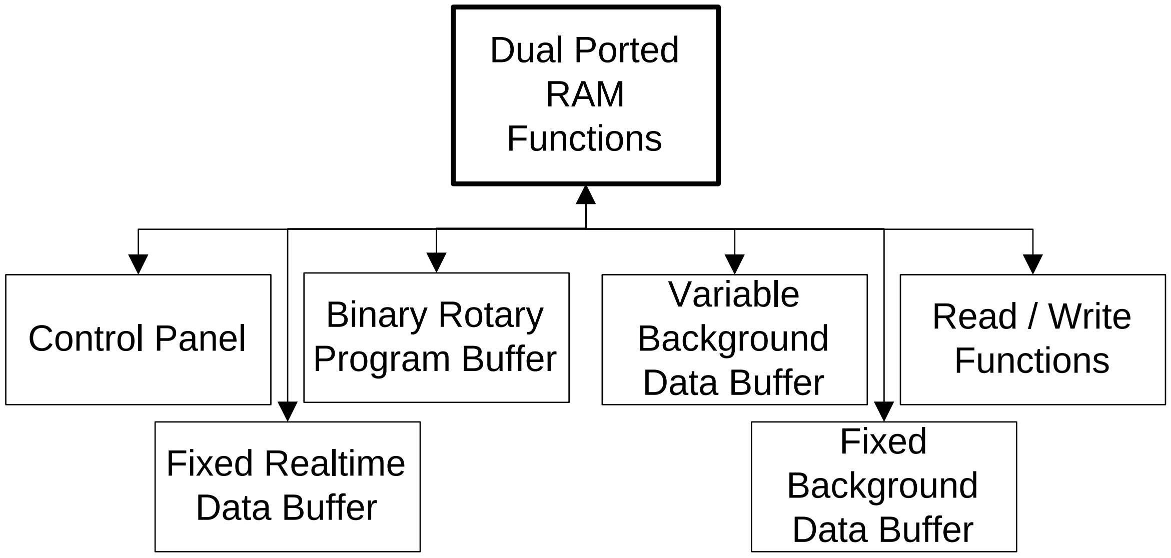
PCOMM32PRO DPR FEATURES

A Global View of the DPR Support Functions

The majority of the functions in this library are Dual Ported RAM (DPR) support functions. Their descriptions in the reference manual have been grouped by functionality as shown below.

Note:

The Automatic DPR Features may be disabled by the PmacSelect() dialog (select device then push the properties button). PmacServer actually performs the update for all applications.


Dual Ported RAM Functions
Control Panel
Binary Rotary Program Buffer
Variable Background Data Buffer
Read / Write Functions
Fixed Realtime Data Buffer
Fixed Background Data Buffer

Dual Ported RAM Support Functionality

Control Panel (Non-Turbo PMACs)

The PMAC provides an area in dual ported RAM that emulates the PMAC's hardware control panel *connector*. When enabled, functions are available to:

1. Jog+, Jog-, Pre-Jog , Home (Motors only).
2. Run, Stop, Step and Feed Hold (Coordinate System only)
3. Assign a feedrate override value for a particular coordinate system.

Fixed Real Time Data buffer

PMAC has an automatic DPR feature, the Fixed Real Time Data Buffer, in which PMAC continually updates a specific area of DPR with a fixed data structure. This data structure is full of meaningful motor (some non-motor information in Non-Turbo PMAC) information and can be accessed by a host application to show positions, velocities etc. in real time. The data in this feature gets updated in PMAC's Real Time Interrupt period.

Fixed Background Data Buffer

This automatic DPR feature is similar to the Fixed Real Time Data Buffer in that a fixed data structure is copied by PMAC into a specific area of the DPR. The difference is that the information is coordinate-system specific, and the information is updated in PMAC's background cycle.

Variable Background Data Buffer

PMAC has the ability to copy up to 128 of it's own registers to the DPR for the host to use. The locations of these 128 registers in PMAC's memory map may be specified by the user (hence the name Variable Background Data Buffer).

Binary Rotary Buffer

This PMAC DPR automatic feature can be used to efficiently download large part programs to PMAC's internal rotary buffer. The routines included to support this feature convert the PMAC ASCII to PMAC Binary before placing the code in the DPR for PMAC to retrieve.

Read/Write Functions

Numeric transfers of 16-and 32-bit wide numbers may be read or written to with this set of PComm32PRO routines. Floating point values are supported for 32-bit transfers. In addition, several helper routines exist for setting individual bits of DPR memory.

Initialization /Shutdown

Note:

The Automatic DPR Features may be disabled by the PmacSelect() dialog (select device then push the properties button). PmacServer actually performs the update for all applications.

Once initialization of communication has been completed (via OpenPmacDevice()), all other DPR functions may be called (of course, only if DPR is present in the system). A complete description of their use and examples are provided in the reference manual.
There is a distinction between "initializing" and "turning on" a DPR feature. Some DPR features require two actions to be taken before they are "running." First you initialize the feature (i.e. for the DPR Rotary buffer call the *PmacDPRRotBufChange()*). Once initiated, the feature is turned off or on with a different function call (i.e. for the DPR Rotary buffer call the *PmacDPRRotBuf()*).
In addition to initialization and shutdown order being important, the order in which a program turns on the DPR features is also critical. What you need to know about the order of turning features on is: If you are using more than one DPR automatic feature, always turn on the DPR Binary Rotary Buffer last. The reason for this is that a call to *PmacDPRRotBuf()* that turns the feature on will open a PMAC program buffer (i.e. the PMAC ASCII command "&1 Open rot"). When a PMAC program buffer is open any attempt to initialize or enable other DPR features will fail (since the driver has to set I-variables to enable a feature, and if a buffer is open the I-variable assignments won't get processed, but rather stored in the buffer).

Memory Organization/Correct Order of Initialization

The order in which the binary rotary buffers and the variable background data buffer are initialized and removed is important to the successful operation of the buffer(s). This is because they are of variable size, and an assumption is made as to the placement of the memory and the sequence of initialization by the library. The order in which they should be initialized is:

1. Binary rotary buffer #0

2. Binary rotary buffer #1
3. Binary rotary buffers (#2 - #7)
4. Variable background data buffer

The following example code demonstrates this:

```
/* 1. Initialize 1st rotary buffer, rotary #0, coordinate system #1 (always)*/
    if (PmacDPRRotBufChange(dwDevice,0,200) < 0){
        MessageBox("***DPR Rotary Buffer #1 Error ***\r\n");
        return;
    }
/*2.Initialize 2nd rotary buffer, coordinate system #2 (always)*/
    if (PmacDPRRotBufChange(dwDevice,1,200) < 0){
        MessageBox("***DPR Rotary Buffer #2 Error ***\r\n");
        return;
    }
// 3. Initialize variable background buffer
    vbg_handle = PmacDPRVarBufInit(dwDevice, 4, VAdarray );
    if ( vbg_handle <0){
        MessageBox("***DPR Background Variable Buffer Error***\r\n");
        return;
    }
// Turn on DPR background data buffer, This MUST BE CALLED
//    BEFORE PmacDPRRotBuf()
    PmacDPRBackground(dwDevice, ON);
// Turn on DPR rotary buffers
    PmacDPRRotBuf(dwDevice, ON);
```

Just like a Last In First Out stack, the order in which the buffers should be removed is:

1. Variable background data buffer
2. Binary rotary buffer #7-#2
3. Binary rotary buffer #1
4. Binary rotary buffer #0

The following shut down procedure is generalized to handle any situation:

```
void closeAll(dwDevice){
      //Shut Down
      int buffnum ;
      while( ( buffnum = PmacDPRBufLast(dwDevice) ) > 0 ){
            if( buffnum == 3 )
                PmacDPRVarBufRemove(dwDevice,vbg_handle);//remove
BG Buf
            if( buffnum == 2 )
                PmacDPRRotBufRemove(dwDevice, 1 );//remove BinRotBuf
#2
            if( buffnum == 1 )
                PmacDPRRotBufRemove(dwDevice, 0 ); // remove BinRotBuf
#1
      }
      if( buffnum < 0 )
            MessageBox("***DPR Buf Remove ERR %d***\r\n", buffnum );
}
```

Here the PmacDPRBufLast() routine is called to determine what type of buffer was placed last in the DPR (3 = Variable Data, 2 = Binary rotary #1, 1 = Binary rotary #0)
The binary rotary and variable background data buffers are placed at the end of the DPR area, allowing the Data Gathering feature to start at the beginning of unreserved memory.

Using the DPR Control Panel (Non-Turbo PMAC)

The PMAC provides an area in the DPR that emulates the PMAC's hardware control panel connector. The PmacDPRControlPanel() function enables/disables the dual ported RAM control panel feature. When enabled, functions are available to:

A) Jog+, Jog-, Pre-Jog , Home (Motors only).
B) Run, Stop, Step and Feed Hold (Coordinate System only)
C) Assign a feedrate override value for a particular coordinate system.

Four necessary conditions for any of the actions above to be realized are:

1. The DPR Control panel has been enabled with a call to PmacDPRControlPanel().
2. The action bit (or flag) for the specified action must first be set to 1 using the corresponding PmacDPRSet*Action*Bit() function, where *Action* is any of those listed in A) and B) above.
3. Any action bits that conflict with the one attempting to be realized must be reset to 0. For example, if the "stop (or abort) bit" has been set and the programmer attempts to run a PMAC program by setting the "run bit," PMAC will not run the program because of the conflict in the actions.
4. The request_bit of the corresponding motor or coordinate system must be set to 1 by use of the PmacDPRSetRequestBit() function.

Action Bits

PMAC reads only action bits after the DPRSetRequestBit() function is called.
None of the action request bits are written to by PMAC. Keeping track of which are set may be facilitated with the PmacDPRGet*Action*Bit() functions.

Request Bits

On the other hand, the request_bit is overwritten by PMAC. PMAC sets this bit to zero when it's finished acknowledging all the requested actions. Because of this fact, the programmer must be sure to wait long

enough for PMAC to process the command before resetting the requested action bit to FALSE (in fact if your program is too fast, PMAC may never see the requested action bit at all).
The request bit can therefore be considered a handshaking bit between the host computer and the PMAC. Host PC sets, PMAC notes the set bit, PMAC performs requested actions, PMAC resets the request bit.
You should build routines that:

1. Set an action bit(s), via PmacDPRSet*Action*Bit()
2. Call the PmacDPRSetRequestBit() function specifying the appropriate Coordinate System or motor for which you want the requested action to occur.
3. Wait until the request bit is reset, via PmacDPRGetRequestBit().
4. Reset the appropriate action bit(s). This makes keeping track of request bits from within your application unnecessary.

An example implementation using "C" is shown below. Here the correct sequence of function calls is shown for jogging motor #1.

```
/* Turn on DPR Control Panel */
   PmacDPRControlPanel(device, TRUE);
/* Set the Jog positive request bit for specified
motor to TRUE */
   PmacDPRSetJogPosBit(device,motor_num, 1);
.
  Other action requests
.
.
/* Set the enable-request bit for the specified motor
to TRUE */
   PmacDPRSetRequestBit(device,motor_num,TRUE);
/* Wait until PMAC processes request */
   while(PmacDPRGetRequestBit(device,motor) &&
i<timeout)
      i++;
/* Reset jog bit to FALSE for housekeeping*/
   PmacDPsetJogPosBit(device,motor,FALSE);
```

Assigning A New Feedrate Override

The necessary conditions for assigning a coordinate system a new feedrate override (also known as "time-base") value are:

1. Ix93, the Time Base Source Address for corresponding coordinate system must be set to it's default value.
2. The DPR Control panel must be enabled with a call to PmacDPRControlPanel().
3. The PmacDPRSetFOValue() must be called to set the feedrate override value for the specified coordinate system.
4. The PmacDPRSetFOEnableBit() must be called to set the feedrate override enable bit for the same coordinate system specified in 3.

The *value* parameter for PmacDPRSetFOValue() may be anything from 1 to 32,767 in units of 1/32,768 msec. The *value* represents the "elapsed time" PMAC uses in its trajectory update equations each servo cycle. If it matches the true time, the trajectories will go at the programmed speeds. If it is greater than the true time, the trajectories will go faster, if it is less, they will go slower. *value* corresponds to values of 0 to 8,388,352 in units of I10 (1/8,388,608 msec). At the default I10 value of 3,713,707, this corresponds to a feedrate override (%) values from 0 to 225.87; for real-time execution (%100) *value* should be 14507.

Feedrate Override Enable Bits

As with the action request bits, the feedrate override enable bits are also not written to by PMAC. PMAC continually writes the feedrate override value specified by PmacDPsetFOValue() when the corresponding enable bit is true. Keeping track of the feedrate override enable bit may be done via the PmacDPRGetFOEnableBit() function, whose return value is the bit value (1 for enabled, 0 for disabled).

Using the DPR Real Time Fixed Data Buffer

Startup/ShutDown

```
BOOL PmacDPRRealTimeEx(DWORD dwDevice, long mask, UINT period, int on
)
void PmacDPRSetRealTimeMotorMask( DWORD dwDevice, long mask )
```

Handshaking

```
BOOL PmacDPRUpdateRealTime( DWORD dwDevice )
```

Data Query functions

Global

```
PmacDPRGetServoTimer()
PmacDPRSysServoError()
PmacDPRSysReEntryError()
PmacDPRSysMemChecksumError()
PmacDPRSysPromChecksumError()
PmacDPRGetGlobalStatus()
```

Coordinate System

```
PmacDPRMotionBufOpen()
PmacDPRRotBufOpen()
PmacDPRGetFeedRateMode()
```

Motor

```
PmacDPRMotorServoStatus()
PmacDPRDataBlock()
PmacDPRPhasedMotor()
PmacDPRMotorEnabled()
BOOL PmacDPRHandwheelEnabled()
BOOL PmacDPROpenLoop()
BOOL PmacDPROnNegativeLimit()
BOOL PmacDPROnPositiveLimit()
```

```
void  PmacDPRSetJogReturn()
MOTION PmacDPRGetMotorMotion()

double PmacDPRGetCommandedPos()
double PmacDPRPosition()
double PmacDPRFollowError()
double PmacDPRGetVel()
void PmacDPRGetMasterPos()
void PmacDPRGetCompensationPos()

DWORD PmacDPRGetPrevDAC()
DWORD PmacDPRGetMoveTime()
```

Note:

The only significant difference between the Turbo and non-Turbo implementation, is the interpretation of the **mask** parameter of PmacDPRRealTimeEx(). Other than that, it is only the Data query methods that differ. Turbo's Realtime data is motor only specific, whereas the non-Turbo contains more generic information.

The PmacDPRRealTimeEx() function enables/disables the DPR real time fixed data buffer. The final parameter **on** will enable this feature if set to 1, otherwise it will disable it. When enabled, all the data query functions above may be used. The **period** parameter in PmacDPRRealTimeEx() specifies how often in servo-cycles PMAC will update the data in this buffer. The data for motor numbers 1- **`mask`** will be updated in DPR where **`mask`** is the second parameter in PmacDPRRealTimeEx () function. The range of motors updated can be modified via the PmacDPRSetRealTimeMotorMask () routine.
The data is refreshed within the DPR when the PmacDPRUpdateRealTime() function is called. Call this before querying data.
The code segment below shows the sequence of calls (An excerpt from the provided PMACTEST application) that should be used to assure that reading of the DPR would not occur while PMAC is writing:
Notice the sequence for handshaking:

1. Refresh the data within the DPR via the PmacDPRUpdateRealtime() routine
2. Call the Data Query functions to read the DPR Real Time Data buffer.

```
void PmacDPRRealTime::OnTimer(UINT nIDEvent)
{
    char buf[256];

// DO HANDSHAKING
    m_pDoc->DPRUpdateRealtime();

    int servotimer = m_pDoc->DPRGetServoTimer();
    sprintf(buf,"%d",servotimer);
    m_ServoCounter.SetWindowText(buf);

    double aDouble = m_pDoc->DPRPosition(iCurrentMotor-1,1);
    sprintf(buf,"%11.1lf",aDouble);
    m_ActualPosition.SetWindowText(buf);

    aDouble = m_pDoc->DPRGetCommandedPos(iCurrentMotor-1,1);
    sprintf(buf,"%11.1lf",aDouble);
    m_CommandedPosition.SetWindowText(buf);

    aDouble = m_pDoc->DPRFollowError(iCurrentMotor-1,1);
    sprintf(buf,"%11.1lf",aDouble);
    m_FollowingError.SetWindowText(buf);

    m_pDoc->DPRGetCompensationPos(iCurrentMotor-1,1,&aDouble);
    sprintf(buf,"%11.1lf",aDouble);
    m_CompensationPosition.SetWindowText(buf);

    aDouble = m_pDoc->DPRGetVel(iCurrentMotor-1,1);
    sprintf(buf,"%11.1lf",aDouble);
    m_Velocity.SetWindowText(buf);

    long aLong = m_pDoc->DPRGetPrevDAC(iCurrentMotor-1);
    sprintf(buf,"%ld",aLong);
    m_PrevDAC.SetWindowText(buf);

    aLong = m_pDoc->DPRGetMoveTime(iCurrentMotor-1);
    sprintf(buf,"%ld",aLong);
    m_MoveTime.SetWindowText(buf);
}
```

Using the DPR Real Time Fixed Data Buffer (Turbo)

Startup/Shutdown

`BOOL PmacDPRRealTimeEx( DWORD dwDevice, long mask, UINT period, int on )`

`void PmacDPRSetRealTimeMotorMask( DWORD dwDevice, long mask )`

Handshaking

`BOOL PmacDPRUpdateRealTime( DWORD dwDevice )`

Data Query functions

Motor

`PmacDPRMotorServoStatus()`

`PmacDPRDataBlock()`

`PmacDPRPhasedMotor()`

`PmacDPRMotorEnabled()`

`BOOL PmacDPRHandwheelEnabled()`

```
BOOL PmacDPROpenLoop()
BOOL PmacDPROnNegativeLimit()
BOOL PmacDPROnPositiveLimit()
void  PmacDPRSetJogReturn()
MOTION PmacDPRGetMotorMotion()

double PmacDPRGetCommandedPos()
double PmacDPRPosition()
double PmacDPRFollowError()
double PmacDPRGetVel()
void PmacDPRGetMasterPos()
void PmacDPRGetCompensationPos()

DWORD PmacDPRGetPrevDAC()
```

Global

```
PmacDPRGetServoTimer()
```

Note:

The only significant difference between the Turbo and non-Turbo implementation, is the interpretation of the **mask** parameter of PmacDPRRealTimeEx(). Other than that, it is only the Data query methods that differ. Turbo's Realtime data is motor only specific, whereas the non-Turbo contains more generic information.

The PmacDPRRealTimeEx() function enables/disables the DPR real time fixed data buffer. The final parameter **on** will enable this feature if set to 1, otherwise it will disable it. When enabled, all the data query functions above may be used. The **period** parameter in PmacDPRRealTimeEx() specifies how often in servo-cycles PMAC will update the data in this buffer. The **mask** parameter is specifies which of the possible 32 motor data sets to update. Bit 0, the least significant bit, enables or disables the first motor by setting it to 1 or 0. Bit 1 enables or disables the second motor etc… The PmacDPRSetRealTimeMotorMask() routine can also be used to modify which motor data sets are being updated after initialization has been done.
Typical usage of these routines is shown below and also is available in the supplied example source code (PmacTest application).

```
void DPRRealTimeTurbo::OnEnablerealtime()
{
    TCHAR buf[255];
  long mask;
  long on = TRUE;
  char *cp;

    m_NumberMotors.GetWindowText(buf,20);
    mask = strtol(buf,&cp,0);
    PmacDPRRealTimeEx (DeviceNum,mask,50,on);

    //  Begin timer
    m_TimerID = SetTimer(
        1,            // timer identifier
        1,                 // 10-second interval
    (TIMERPROC) NULL);     // no timer callback
}

void DPRRealTimeTurbo::OnDisablerealtime()
{
    long on = FALSE;
    PmacDPRRealTimeEx(DeviceNum,0,0,on);
    KillTimer(m_TimerID);
}
void PmacDPRRealTime::OnTimer(UINT nIDEvent)
{
    DWORD dwDevice = 0;
    Double units = 1;
    Long motor = iCurrentMotor-1;

    m_pDoc->DPRDoRealTimeHandshake();

    int servotimer = PmacDPRGetServoTimer(dwDevice);
    sprintf(buf,"%d",servotimer);
    m_ServoCounter.SetWindowText(buf);

    double aDouble = PmacDPRPosition (dwDevice,motor,units);
    sprintf(buf,"%11.1lf",aDouble);
    m_ActualPosition.SetWindowText(buf);

    aDouble = PmacDPRGetCommandedPos (dwDevice,motor,units);
    sprintf(buf,"%11.1lf",aDouble);
    m_CommandedPosition.SetWindowText(buf);

    aDouble = PmacDPRFollowError (dwDevice, motor,units);
    sprintf(buf,"%11.1lf",aDouble);
    m_FollowingError.SetWindowText(buf);

   aDouble = PmacDPRGetBiasPos (dwDevice, motor,units);
    sprintf(buf,"%11.1lf",aDouble);
    m_BiasPos.SetWindowText(buf);

    aDouble = PmacDPRGetVel (dwDevice,motor,3072.0);
    sprintf(buf,"%11.1lf",aDouble);
    m_Velocity.SetWindowText(buf);

    long aLong = PmacDPRGetPrevDAC (dwDevice,motor);
    sprintf(buf,"%ld",aLong);
    m_PrevDAC.SetWindowText(buf);

    CDialog::OnTimer(nIDEvent);
}
```

Using the DPR Background Fixed Data Buffer

Startup/Shutdown and Handshaking Functions

```
BOOL  PmacDPRBackground(DWORD dwDevice, long on_off);
BOOL  PmacDPRSetBackground(DWORD dwDevice);
void PmacDPRSetMotors(DWORD dwDevice,UINT numberOfMotors);
```

Data Query Functions

Motor Specific

```
double PmacDPRCommanded(DWORD dwDevice,long crd,char maxchar);

double PmacDPRGetVel(DWORD dwDevice,long motor,double units);

double PmacDPRVectorVelocity(DWORD dwDevice,long num,long motor[],
double units[]);

BOOL PmacDPRAmpEnabled(DWORD dwDevice,long motor);

BOOL PmacDPRWarnFError(DWORD dwDevice,long motor);

BOOL PmacDPRFatalFError(DWORD dwDevice,long motor);

BOOL PmacDPRAmpFault(DWORD dwDevice,long motor);

BOOL PmacDPROnPositionLimit(DWORD dwDevice,long motor);

BOOL PmacDPRHomeComplete(DWORD dwDevice,long motor);

BOOL PmacDPRInposition(DWORD dwDevice,long motor);

double PmacDPRGetTargetPos(DWORD dwDevice,long motor, double
posscale);

double  PmacDPRGetBiasPos(DWORD dwDevice,long motor, double posscale);
```

Coordinate System Specific

```
long PmacDPRPe(DWORD dwDevice,long cs);

BOOL PmacDPRRotBufFull(DWORD dwDevice,long crd);

BOOL PmacDPRSysInposition(DWORD dwDevice,long crd);

BOOL PmacDPRSysWarnFError(DWORD dwDevice,long crd);

BOOL PmacDPRSysFatalFError(DWORD dwDevice,long crd);

BOOL PmacDPRSysRunTimeError(DWORD dwDevice,long crd);

BOOL PmacDPRSysCircleRadError(DWORD dwDevice,long crd);

BOOL PmacDPRSysAmpFaultError(DWORD dwDevice,long crd);

BOOL PmacDPRProgRunning(DWORD dwDevice,long crd);

BOOL PmacDPRProgStepping(DWORD dwDevice,long crd);

BOOL PmacDPRProgContMotion(DWORD dwDevice,long crd);

BOOL PmacDPRProgContRequest(DWORD dwDevice,long crd);

long PmacDPRProgRemaining(DWORD dwDevice,long crd);

long PmacDPRTimeRemInMove(DWORD dwDevice,long cs);
long PmacDPRTimeRemInTATS(DWORD dwDevice,long cs);
```

Global

```
PmacDPRSysServoError()
PmacDPRSysReEntryError()
PmacDPRSysMemChecksumError()
PmacDPRSysPromChecksumError()
PmacDPRGetGlobalStatus()
```

Logical Query Functions

```
MOTIONMODE  PmacDPRGetMotionMode(DWORD dwDevice,long cs);
PROGRAM  PmacDPRGetProgramMode(DWORD dwDevice,long cs);
enum MOTION  PmacDPRGetMotorMotion(DWORD dwDevice,long motor);
```

To enable this feature, call the PmacDPRBackground() routine with the on_off parameter set to a non-zero value. To retrieve the information, use the provided DPR Background Fixed Data buffer query functions. Information for motors/coordinate systems 1-*n* will be updated, where *n* is the second parameter of the PmacDPRSetMotors() function. You may call DPRSetMotors() before or after PmacDPRBackground().
When enabled, all the query functions above may be used to:

- Query PMAC's control panel, thumbwheel and machine I/O connector status.
- Get motor target, bias and commanded position registers. Also query a motor's status word.
- Query a coordinate systems status word, program status, program remaining, time remaining in move and acceleration and finally the coordinate systems currently executing line.

Once this function is enabled, call the DPRSetBackground() function to assure that the data query functions get fresh data. All data query functions have this function within it relieving you from dealing with the handshaking details. A typical sequence of function calls to data would be:

```
PmacDPRSetMotors(p,4);// Update data for motors/cs 1 - 4
p  =  PMAC_DEVICE_NUMBER;
while(!DONE){
        printf("Target Position: %11.1lf\n",PmacDPRGetTargetPos(p, mtrcrd,1.0/(96.0*32.0)));
        printf("Bias Position: %11.1lf\n",PmacDPRGetBiasPos(p,mtrcrd,1.0/(96.0*32.0)));
        aDouble=PmacDPRCommanded(p,mtrcrd,'A');
        printf("Commanded Position A: %11.4lf\n",aDouble);
        aDouble=PmacDPRCommanded(p,mtrcrd,'B');
        printf("PmacCommanded Position B: %11.4lf\n",aDouble);
        aDouble=PmacDPRCommanded(p,mtrcrd,'C');
        printf("Commanded Position C: %11.4lf\n",aDouble);
        .
        .
        printf("WFE:%d\n",PmacDPRWarnFError(p,mtrcrd));
        printf("FFE:%d\n",PmacDPRFatalFError(p,mtrcrd));
        printf("AMP FAULT:%d\n",PmacDPRAmpFault(p,mtrcrd));
        printf("POSITION LIMIT:%d\n",PmacDPROnPositionLimit(p,mtrcrd));
        printf("HOME COMPLETE:%d\n",PmacDPRHomeComplete(p,mtrcrd));
        printf("MOTOR IN POSITION:%d\n",PmacDPRInposition(p,mtrcrd));
}
```

Using the DPR Background Fixed Data Buffer (Turbo)

Startup/Shutdown and Handshaking Functions

```
BOOL PmacDPRBackgroundEx(DWORD dwDevice,int on,UINT period,UINT crd);
```

Data Query Functions

Motor Specific

```
double PmacDPRCommanded(DWORD dwDevice,long crd,char maxchar);
double PmacDPRGetTargetPos(DWORD dwDevice,long motor, double
posscale);
```

Coordinate System Specific

```
double PmacDPRGetFeedRateMode(DWORD dwDevice,int csn, BOOL *mode)
BOOL PmacDPRMotionBufOpen( DWORD dwDevice)
long PmacDPRPe(DWORD dwDevice,long cs);
BOOL PmacDPRRotBufFull(DWORD dwDevice,long crd);
BOOL PmacDPRRotBufOpen( DWORD dwDevice )
BOOL PmacDPRSysInposition(DWORD dwDevice,long crd);
BOOL PmacDPRSysWarnFError(DWORD dwDevice,long crd);
BOOL PmacDPRSysFatalFError(DWORD dwDevice,long crd);
BOOL PmacDPRSysRunTimeError(DWORD dwDevice,long crd);
BOOL PmacDPRSysCircleRadError(DWORD dwDevice,long crd);
BOOL PmacDPRSysAmpFaultError(DWORD dwDevice,long crd);
BOOL PmacDPRProgRunning(DWORD dwDevice,long crd);
BOOL PmacDPRProgStepping(DWORD dwDevice,long crd);
BOOL PmacDPRProgContMotion(DWORD dwDevice,long crd);
BOOL PmacDPRProgContRequest(DWORD dwDevice,long crd);
long PmacDPRProgRemaining(DWORD dwDevice,long crd);
long PmacDPRTimeRemInMove(DWORD dwDevice,long cs);
long PmacDPRTimeRemInTATS(DWORD dwDevice,long cs);
```

Global

```
BOOL PmacDPRSysServoError( DWORD dwDevice )
BOOL PmacDPRSysReEntryError( DWORD dwDevice )
BOOL PmacDPRSysMemChecksumError( DWORD dwDevice )
BOOL PmacDPRSysPromChecksumError( DWORD dwDevice )
void PmacDPRGetGlobalStatus(DWORD dwDevice,VOID *gstatus)
```

Logical Query Functions

```
MOTIONMODE PmacDPRGetMotionMode(DWORD dwDevice,long cs);
PROGRAM PmacDPRGetProgramMode(DWORD dwDevice,long cs);
```

To enable this feature, call the PmacDPRBackgroundEx() routine with the **on** parameter set to a non-zero value, the **period** argument set anywhere from 1 to 255 (servo periods) and the **crd** parameter set from 1 – 8. To retrieve the information, use the provided DPR Background Fixed Data buffer query functions. Information for motors/coordinate systems 1**- crd** will be updated.
When enabled, all the data query functions above may be used.
Typical usage of these routines is shown below and also is available in the supplied example source code (PmacTest application).

```
void BackgroundFixedTurbo::OnDisablebgbuf()
{
    PmacDPRBackgroundEx (dwDevice,FALSE,0,0);
    KillTimer(m_TimerID);
    m_bg_buf_enabled = FALSE;
}

void BackgroundFixedTurbo::OnEnablebgbuf()
{
    TCHAR buf[255];

    m_text_I59.GetWindowText(buf,255);
    USHORT temp = atoi(buf);

    if(!PmacDPRBackgroundEx (dwDevice, TRUE,100,temp))
        ::MessageBox(m_hWnd,"Unable to Initialize Background
Buffer","Information",MB_OK|MB_ICONINFORMATION);
    else{
        m_bg_buf_enabled = TRUE;
//  Begin timer
        m_TimerID = SetTimer(
            1,            // timer identifier
            5,                  // 10-second interval
        (TIMERPROC) NULL);     // no timer callback
    }
}

void BackgroundFixedTurbo::OnTimer(UINT nIDEvent)
{
    TCHAR buf[300];
    double my_double;
  BOOL myBool;

    if(m_bg_buf_enabled){

        my_double = PmacDPRCommanded(dwDevice,m_SelectedCS,'A');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedA.SetWindowText(buf);

        my_double= PmacDPRCommanded(dwDevice,m_SelectedCS,'B');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedB.SetWindowText(buf);

        my_double= PmacDPRCommanded(dwDevice,m_SelectedCS,'C');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedC.SetWindowText(buf);

        my_double=PmacDPRCommanded(dwDevice,m_SelectedCS,'U');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedU.SetWindowText(buf);

        my_double=PmacDPRCommanded(dwDevice,m_SelectedCS,'V');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedV.SetWindowText(buf);

        my_double=PmacDPRCommanded(dwDevice,m_SelectedCS,'W');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedW.SetWindowText(buf);

        my_double=PmacDPRCommanded(dwDevice,m_SelectedCS,'X');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedX.SetWindowText(buf);

        my_double=PmacDPRCommanded(dwDevice,m_SelectedCS,'Y');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedY.SetWindowText(buf);

        my_double=PmacDPRCommanded(dwDevice,m_SelectedCS,'Z');
        sprintf(buf,"%11.4lf",my_double);
        m_textCommandedZ.SetWindowText(buf);
        wsprintf(buf,"%d\r\n", PmacDPRProgRemaining (dwDevice,
m_SelectedCS));
        m_textProgRemaining.SetWindowText(buf);

        wsprintf(buf,"0x%04lx\r\n", PmacDPRPe(dwDevice,
m_SelectedCS));
        m_textOffsetAddress.SetWindowText(buf);

        wsprintf(buf,"%7ld\r\n", PmacDPRTimeRemInMove(dwDevice,
m_SelectedCS));
        m_textTIM.SetWindowText(buf);

        wsprintf(buf,"%7ld\r\n", PmacDPRTimeRemInTATS(dwDevice,
m_SelectedCS));
        m_textTATS.SetWindowText(buf);

    my_double= PmacDPRGetFeedRateMode
(dwDevice,m_SelectedCS,&myBool);
        sprintf(buf,"%11.4lf", my_double);
        m_FeedRate.SetWindowText(buf);

  }
    CDialog::OnTimer(nIDEvent);
}
```

Using the DPR Variable Background Read/Write Data Buffer

Variable Read Data Buffer

The variable background data buffer can be used to have PMAC copy any of it's own internal registers into the DPR area. The registers copied are specified by using the startup functions discussed below. Up to 128 registers can be copied.

These routines may work in a multitasking environment, such as Windows. That is, multiple applications may use the Variable Background Data Buffer (VBGDB) simultaneously (for Prom versions 1.15G and above only, earlier proms are single user only), although the combined number of registers copied still remains at 128.

Startup/Shutdown

long PmacDPRVarBufInit(DWORD dwDevice, long num_entries, long *addrarray);

PmacDPRVarBufInit() is used to initialize or append the VBGDB. If all goes well, a non-zero "handle" is returned to the caller. The handle returned is necessary for all data querying functions, and therefore should be stored for safekeeping. A return value of FALSE indicates initialization failed.

See Memory Organization/Correct Order of Initialization for a discussion on how memory is managed in the DPR, and also the order in which buffers should be initialized and removed. This buffer should be added after all binary rotary buffers have been initialized, and removed before the binary rotary buffers are deleted.

You must specify the number of entries to be updated by PMAC (**num_entries**) and all PMAC addresses to be copied into the DPR (the **addrarray** parameter, a long array of **num_entries** size). PMAC addresses are specified using an array of long integers. The most significant word of each long (upper 16 bits) specifies the word type. A value of 0, 1, 2 and 4 corresponds to Y, Long, X, and SPECIAL respectively. For Y, Long and X entries the least significant word specifies the actual PMAC address to be copied.

Take for example the following define statements and the initialization of a long integer array.

```
#define MTR1_ACTULPOS     0x00010028 /* PMAC Address D:$0028 */

#define MTR1_ACTULVEL     0x00020033 /* PMAC Address X:$0033 */

#define MTR1_DACOUT                 0x00020045 /* PMAC Address X:$0045 */
```

long myaddrarr[3]={MTR1_DACOUT,MTR1_ACTULPOS,MTR1_ACTULVEL};

If myaddrarr is used as the last parameter in the PmacDPRVarBufInit() function the second parameter, **num_entries**, which is the number of elements in the long array would have to be 3. Because only 128 entries may be updated, you may want to call PmacDPRGetVBGTotalEntries() to determine how many entries are already in use before calling PmacDPRVarBufInit(). Although PmacDPRVarBufInit() will also check the available space and return an error if the request is too large. The PmacDPRBackGroundVar() function enables/disables this feature. A return value 1 means successful, 0 failure.

Once initialized, a different set of PMAC registers to be copied to DPR may be specified by using the PmacDPRVarBufChange() function. This function works exactly as PmacDPRVarBufInit() except that it has one additional parameter, the handle to the already existing background data variable buffer

Shutting down the VBGDB is done by calling the PmacDPRVarBufRemove() function. This function de-allocates memory within PComm32PRO, and should always be called before any application using the VBGDB terminates. Lastly, a call to PmacDPRBackGroundVar() to disable the PMAC updating of this buffer is suggested.

Data Query Function Description

Once the variable buffer is initialized and enabled, data may be retrieved by use of the PmacDPRVarBufRead() function. The entry number, **entry_num**, specifies to PComm which of the elements in the address array (**addrarray)** passed to it, the calling function desires. In the example above, to retrieve motor 1 actual position, the second entry in the address array, the programmer would set **entry_num** to 1 (i.e. the first entry would be 0). The parameter **long_2** is a long pointer that should point to a memory element at least two long integers in size. One could use the following declared element for the **long_2** parameter:

long long_array[2];

A PMAC Short word is, for example, an X or Y location (24 bits copied into 32 DPR bits). A Long word would be an L or D word (48 bits copied into 64 DPR bits).

If the data is ready when this function is called, a value of TRUE is returned, and the data is placed in the **long_2** parameter. If the requested entry is a PMAC short word, only the first element in the long array is filled (i.e. long_array[0]) and the second element is not relevant. On the other hand, if a PMAC long word or SPECIAL (PLCC Function block) is requested, both entries in the long array are filled with data. PMAC long words (48 bits) occupy two long words in DPR (64 bits, 2 "entries"). PmacDPRLFixed() or PmacDPRFloat() can be used to convert the long array to a meaningful numeric value. For example, if three entries are to be updated and they are in this order:

1. Motor 1's DAC output (X:$0045)
2. Motor 1's actual position (D:$0028)
3. Motor 1's Actual velocity (X:$0033)

Then entry_num=0 corresponds to motor 1's DAC output, entry_num = 1 correspond to the actual position registers least and most significant words and finally entry_num = 2 corresponds to motor 1's actual velocity register. The following demonstration source illustrates this process.

```
/* Get the dual word data */

        long data[2];

While(1){

        /* Get second entry*/

        if(PmacDPRVarBufRead(p,myVBGBHandle,1,data)){

                /* Convert to encoder counts (Assuming I108 = 96)*/

                MotorPosition = PmacDPRLFixed(data,1.0/(96.0*32.0));

                cprintf("\n\n Actual motor position after conversion
        =%lf\r\n",MotorPosition);
        }
}
```

All the handshaking necessary for this buffer is contained within DPRVarBufRead(). Simply poll the return value of this function to determine whether or not the data is ready.
An extremely efficient function, **PmacDPRVarBufReadEx()**, allows you to grab all of the data within the VBGDB in one call. Like its counterpart, PmacDPRVarBufRead(), it handles the necessary handshaking required. The only significant difference is the size of the long array passed to it. The data array should be at least (2 * num_entries) large. Also, PmacDPRVarBufReadEx() does not have a parameter specifying which entry to get, as it will get them all.

Note:

If you are not using any of the Binary Rotary Buffers, set dprVarStart, the initialization file parameter, to at most 0xDFFE. This way the last entry in your variable background data buffer will be updated.

PmacDPRGetVBGStartAddr()
Use this to attain the PMAC address of the start of the variable data buffer. This address also indicates the end of the free user memory.
PmacDPRBufLast()

This function determines which buffer was the last buffer entered in DPR.
Return Value:

0 = No Buffers in DPR
1 = Binary Rotary Buffer #1
2 = Binary Rotary Buffer #2
3 = Background Variable Buffer

Use this function to decide which buffer (either a binary rotary or the variable background data buffer) can be initialized or removed. The sequence for the binary rotary and variable background data buffers is like a last in first out stack. The order of initialization must be: 1. Binary rotary buffer 0, 2. Binary rotary buffer 1, 3. Variable background data buffer.
PmacDPRGetVBGServoTimer()
Returns the value of PMAC's servo timer
DPRGetVBGNumEntries()
Returns the number of items being updated in the buffer by the calling application. This number is the same as that passed to the function PmacDPRVarBufInit().
PmacDPRGetVBGTotalEntries()
Returns the number of items being updated in the buffer by all applications. Should never exceed 128.
DPRGetVBGAddress()
This function returns an entry within the address table initialized by PmacDPRVarBufInit(). For the first entry set entry_num to 0, the second set entry_num to 1 etc. PMAC reads this address table to determine what to copy into DPR memory. Typically not used by programmers.
DPRGetVBGDataOffset()
Returns the offset, in bytes, from the beginning of DPR that the data for this applications VBGDB begins. Typically not used by programmers.
DPRGetVBGAddrOffset()
Returns the offset, in bytes, from the beginning of DPR that the address array for this application's VBGDB begins. Divide by 4 to get the PMAC offset (from beginning of DPR 0xD000). Typically not used by programmers.

Variable Write Data Buffer

`long PmacDPRWriteBufferEx(DWORD dwDevice, long num_entries, struct VBGWFormat *the_data);`

This feature is available for PROM Version 1.15G and above.

The variable background write buffer can be used to have PMAC transfer up to 32 PMAC long or short words from DPR to its own internal memory without using the DPR ASCII feature. The data to be written into PMAC's memory is first placed in an array of VBGWFormat structures (definition shown below).

```
struct VBGWFormat{
      long type_addr;
      long data1;
      long data2;
};
```

The upper 16 bits of type_addr element specifies the type of data and the lower 16 bits specify the address to which data1 and data2 should be written.

The types of data that may be specified are as follows.

Bits 0-2	0/1/2/4 for Y/L/X/SPECIAL memory respectively.
Bits 3-7	Width = 1,4,8,12,16,20 (a value of 0 is 24 bit variable).
Bits 8-12	Offset = 0..23.
Bits 13-15	Special type.

The way in which one uses data1, and data2 is not intuitive. For Y or X memory data1 element will be used for specifying the 24 bits of data. The most significant byte of data1 and all of data 2 are irrelevant in this case. For L data, PMAC 48-bit word, data1 should hold the first 32 bits and data2 should hold the most significant 16 bits, leaving the most significant 16 bits of data2 irrelevant.

TWS is not yet implemented.

PMAC addresses are specified with an array of long integers. The most significant word of each long (upper 16 bits) specifies the word type. A value of 0, 1, 2 and 4 corresponds to Y, Long, X, and SPECIAL, respectively. For Y, Long and X entries, the least significant word specifies the actual PMAC address to be copied.

Using the DPR Binary Rotary Motion Program Buffer

<u>Startup/Shutdown functions</u>

`PmacDPRRotBufChange()`

`PmacDPRRotBufRemove()`

`PmacDPRRotBufClr()`

`PmacDPRRotBuf()`

`PmacDPRBufLast()`

<u>Conversion & Transfer functions</u>

`PmacDPRAsciiFileToRot()`

`PmacDPRBinaryFileToRot()`

```
PmacDPRAsciiStrToRot()
PmacDPRAsciiStrToRotEx()
PmacDPRBinaryToRot()
PmacDPRAsciiToBinaryFile()
PmacDPRAsciiStrToBinaryFile()
```

PmacDPRRotBufChange()

The PmacDPRRotBufChange() function is used to initialize the DPR Binary Rotary motion program buffers. Presently, there may be up to eight rotary buffers, buffer #0 through buffer #7 (All Turbo PMAC support eight buffers; however, non-Turbo PMACs with firmware earlier than 1.16B will have only two). Buffers #0, and #1 correspond to coordinate system 1 and 2, respectively. The binary rotary buffer you wish to initialize must be the last variable-sized buffer to be created. Therefore, if you wish to use three rotary buffers, begin with #0, then #1 and finally #2.

The size of a particular buffer is also specified with this function. Note that using PmacDPRRotBufChange() will not cause the change in size to be persistent when resetting the operating system (registry is not changed). Note also that you should set I57 to 0 before changing the buffer size, and the buffer being changed should be the one last initialized. If size is set to zero, this function is equivalent to PmacDPRRotBufRemove().

Typically, the size of the rotary buffer in PMAC is half that allocated in the DP Ram. Two long words in the DP Ram (64 bits), is mapped into one long PMAC word (48 bits).

The value assigned to size, which is in units of 32 bits, specifies how many instructions will fit into the buffer. Ultimately, each ASCII instruction is transformed into a 64-bit code (from which PMAC will transform into a 48-bit operation code). A single instruction consists of a program command and it's argument. For example, "X100" is a single instruction and "X1000Y1000Z1000" is three instructions. A simple formula to determine the bufsize as a function of the maximum number of buffered program statements is:

buffer_size = (number_of_motors_in_cs) * (num_lines) * (2);

The Rotary buffer size must be at least six words long. If it is too large, then it may run into the gather buffer

Call PmacDPRRotBuf() to enable the Binary Rotary Buffer after it has been initialized.

PmacDPRRotBufRemove()

Use this function to remove any previously initialized rotary buffer.

PmacDPRRotBufClr()

This function will clear a Binary Rotary buffer in DPR (i.e. remove all entries). You should disable the Binary Rotary Buffer first by calling PmacDPRRotBuf().

PmacDPRRotBuf()

Once initialized with PmacDPRRotBufChange(), the DPRotBuf() function can be used to enable or disable the rotary buffer (if argument **onoff** = 0 then disable else enable). The return value is a Boolean, TRUE for enabled, FALSE for disabled.

PmacDPRBufLast()

This function determines which buffer is the last variable-sized buffer in DPR.

Return Value:

From 1.16B and beyond

9 = Background Variable Buffer

8 = Binary Rotary Buffer #7

7 = Binary Rotary Buffer #6

6 = Binary Rotary Buffer #5

5 = Binary Rotary Buffer #2

4 = Binary Rotary Buffer #4

3 = Binary Rotary Buffer #3

2 = Binary Rotary Buffer #2

1 = Binary Rotary Buffer #1

0 = No Buffers in DPR

Before 1.16B

3 = Background Variable Buffer

2 = Binary Rotary Buffer #2

1 = Binary Rotary Buffer #1

0 = No Buffers in DPR

Use this function to decide which buffer (either a binary rotary or the variable background data buffer) can be initialized or removed. The sequence for the binary rotary and variable background data buffers is like a last in first out stack. The order of initialization must be:

1. Binary rotary buffer 0
2. 2. Binary rotary buffer 1,
3. 3. Variable background data buffer

Downloading "Stuffing" functions

PMAC ASCII program command strings either a) Created on the fly or b) Retrieved from a file, are converted to binary and sent to the DPR rotary buffer via the PmacDPRAsciiStrToRot() or PmacDPRAsciiStrToRotEx() functions.

Several things must be done before a rotary program can run:

1. Configure a coordinate system(s) in PMAC.
2. Close the servo loops, and make sure all motors are ready to run a motion program (i.e. there not jogging homing etc.).
3. Initialize the rotary buffer in PMAC's internal memory, using the DEF ROT x command. You must start with the highest numbered coordinate system.
4. Initialize a rotary program buffer for the corresponding coordinate system (i.e. call `PmacDPRRotBufChange()`).
5. Open PMAC's rotary buffer. (“Open Rot” for non-Turbo, or “Open Bin Rot” for Turbo).
6. Begin execution of the rotary buffer (i.e. "B0R").

Items 1 and 2 may have to be done only once for a given application.

Return Codes Possible

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"

IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons, when this occurs the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

If you get something other than a DprBufBsy, DprOk, or DprEOF, I'd flag the user of the error. In this case the error is a conversion issue (converting to ASCII to BINARY).

PROGRAMMER'S REFERENCE

INTRODUCTION TO PROGRAMMER'S REFERENCE

The Programmers Reference of PComm32PRO details all of the PMAC library functions in groups of similar functionality. The description of each function includes the syntax, arguments, and possible return values. The groups are ordered as follows:

1. Initialization, Shutdown and Configuration Functions
2. ASCII Communication Functions
3. Downloading Functions
4. DPR Control Panel Functions
5. DPR Real Time Fixed Data Buffer
6. DPR Variable Background Data Buffer Functions
7. DPR Binary Rotary Buffer Functions
8. DPR Numeric Read / Write Functions
9. Interrupt Functions
10. Variable Functions
11. Data Types and Structures

Important Information About Method of Communication Being Used by PComm32PRO

There are three methods by which PComm32PRO may be used to communicate to PMAC, over the Bus, Dual Ported Ram, or the Serial Port. Immediately after initialization (after a call to OpenPmacDevice()) the method used depends on what is stored in the system registry (either BUS or SERIAL). To change the method used at startup, call the PmacConfigure() function so that the change is saved in the registry. Alternatively, if your operating system has the ability to configure drivers (i.e. in the control panel or the multimedia palette) go there and select the PMAC driver for configuration.

Maximum Number of Response Buffer Characters

Many of the ASCII Communication functions use an argument by the name of **maxchar**. The **maxchar** specifies the maximum number of characters that will be placed within a response buffer (a character string used as a parameter to the same function) and should therefore never be larger than the size of the response buffer. If PMAC's response be larger than the value specified by maxchar the remainder of the response will remain in PMAC's queue.

INITIALIZATION, SHUTDOWN AND CONFIGURATION FUNCTIONS

OpenPmacDevice()

```
BOOL OpenPmacDevice(DWORD dwDevice);
```

This function opens a channel for your program to use the PMAC driver.
In order for this function to succeed, the PMAC Win32 Driver must be previously installed in the operating system. PMAC(*dwDevice*) must be registered in the environment. Then the system registry will contain the location and configuration of the PMAC specified by *dwDevice*. OpenPmacDevice looks to the registry for this information. The registry values are located in HKLM/System/CurrentControlSet/Services/Pmac/Device(*dwDevice*).
Every *OpenPmacDevice()* should be paired with a call to *ClosePmacDevice()* to release the resources used by the driver.

Arguments

dwDevice	Device number to open.

Return Value

TRUE if success.

See Also

ClosePmacDevice()

ClosePmacDevice()

```
BOOL ClosePmacDevice(DWORD dwDevice);
```

This function closes the channel from your program to the PMAC driver.

Arguments

dwDevice	Device number to close.

Return Value

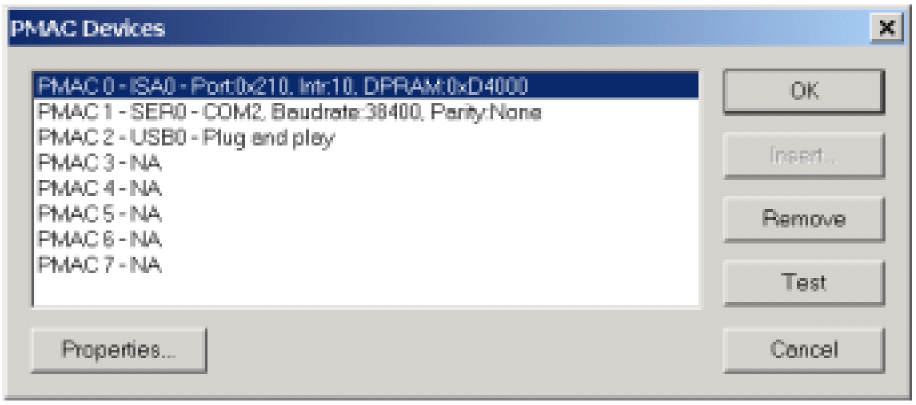
TRUE if success.

See Also

OpenPmacDevice()

PmacSelect()

```
long PmacSelect( HWND hwnd );
```



PMAC 0 - ISA0 - Port:0x210, Intr:10, DPRAM:0xD4000
PMAC 1 - SER0 - COM2, Baudrate:38400, Parity:None
PMAC 2 - USB0 - Plug and play
PMAC 3 - NA
PMAC 4 - NA
PMAC 5 - NA
PMAC 6 - NA
PMAC 7 - NA

Provides a way to select and configure currently installed PMAC Devices. A dialog box is displayed, as shown, to allow selection and configuration of all possible PMAC devices. PMAC devices available are those whose driver has been installed. Typically this is used to allow end users of an application to pick and choose from several PMAC devices in a PC.

Arguments

hwnd	Handle to parent window for device configuration dialog.

Return Value:

```
>= 0 and <= 7 : Device selected
-1 or FFFFFFFF : User aborted with Cancel button.
```

LockPmac()

```
void LockPmac(DWORD dwDevice);
```

You may find that you need to communicate to PMAC (via ASCII communication), un-interrupted by other threads or applications, from time to time. Use LockPmac() and ReleasePmac() routines for these instances. Most ASCII communication routines are thread safe, meaning they employ the usage of LockPmac() and ReleasePmac() internally. However, there are a few that are not, namely: (PmacSendLine(), PmacGetLine() and PmacSendChar()).
The LockPmac() routine asks the operating system if the specified PMAC resource is available. If it is, the operating system will now restrict usage to the PMAC to that calling thread. If it is not available, the operating system will put the calling thread in a wait state until it does become available.
The ReleasePmac() routine releases the PMAC resource for other threads to access. It is critical to always have a release for every LockPmac() call. You may nest LockPmac()-ReleasePmac() sets.

Note:

If your application suddenly locks up, try to recall when and where you call LockPmac() and ReleasePmac(). Probably you did not release as many times as you locked.

Arguments

dwDevice	Device number.

Return Value

None

See Also

ReleasePmac(), “Thread Safe ASCII Communications”

ReleasePmac()

```
void ReleasePmac(DWORD dwDevice); See LockPmac() for description.
```

Arguments

dwDevice	Device number.

Return Value

None

See Also

LockPmac(),“Thread Safe ASCII Communications”

PmacSetCriticalVars()

```
BOOL PmacSetCriticalIVars(DWORD dwDevice);
```

This routine optimizes PMAC’s I-variables for the most efficient and robust communication. The following variables are set:

- I3=2, PMAC sends <CR> followed by an <ACK> after every valid response.
- I6=1, This parameter reports how PMAC reports errors in command lines. When I6=1, the form of the error message is <BELL>{error message}.
- I63=1, Pmacfush sends ^X to PMAC. With I63=1 PMAC returns ^X back to acknowledge the flush process.
- I64=1, Each Unsolicited response is tagged with a ^B in front so that the user can retrieve the message and display it.

Note:

These variables are set automatically during the OpenPmacDevice() routine. For older firmware revisions for non-Turbo PMACs, i.e, 1.16D and older version did not support I63 and I64. Therefore, it is strongly recommended to upgrade older firmware versions to newer ones; otherwise PmacFlush() will time out during every GetResponseEx() and PMAC response will be very sluggish.

Arguments

dwDevice	Device number.

Return Value

TRUE if success. Return value indicates whether or not the new firmware variables have been enforced.

PmacAbortDPRTest()

```
void PmacAbortDPRTest(DWORD dwDevice);
```

Stops the dual ported RAM test function initiated by the PmacDPRTest() function.

Arguments

dwDevice	Device number to stop test for.

Return Value

TRUE if success.

See Also
PmacDPRTest()

PmacConfigure()

```
BOOL   PmacConfigure(HWND hwnd,DWORD dwDevice);
```

This function causes the PComm32PRO Driver's configuration dialog to be presented to the user for driver configuration. If the device is open, it will be closed by this function. This function can be used to modify the many attributes associated with talking to a particular PMAC. These attributes are stored in the system's registry so that they will persist when the system recycles power or reboots.

Arguments

hwnd	Handle to parent window for device configuration dialog.
dwDevice	Device number to configure.

Return Value
TRUE if success.

PmacGetDpramAvailable()

```
BOOL   PmacGetDpramAvailable(DWORD dwDevice);
```

This function determines if dual ported RAM is available for use by your application. If the PMAC controller has dual ported RAM available and it was successfully initialized by the driver, this function will return TRUE.

Arguments

dwDevice	Device number.

Return Value
TRUE if Dual Ported RAM is available.

PmacInBootStrapMode()

```
BOOL PmacInBootStrapMode(DWORD dwDevice);
```

This function returns TRUE if the PMAC controller is in a mode known as BOOTSTRAP. It is in this mode that PMACs equipped with flash ram will be able to accept a new firmware file. This mode is usually caused by a jumper being placed on the controller (i.e. E3 on PMAC2, E51 on PMAC).
Since PMACs command set is totally different when in this mode, it makes sense at certain times to check for this. For example, this function is used in Delta Tau's Executive program when it first starts up. If the board is in BOOTSTRAP mode, it changes the file menu option to allow for firmware download.

Arguments

dwDevice	Device number.

Return Value
TRUE if the PMAC controller is in BOOTSTRAP mode.

See Also

`PmacDownloadFirmwareFile()`

PmacDPRTest()

```
LONG PmacDPRTest(DWORD dwDevice, DPRTESTMSGPROC msgp, DPRTESTPROGRESS prgp);
```

This function provides a method for Dual Ported RAM testing. Both progress and message callback functions are provided (See *Data Types, Structures, Callbacks, and Constants* in this manual for info on callback prototypes and descriptions).

Note:

This function will also set PMAC parameters I2,I5,I47,I48 and I49 in order to prevent the dual ported RAM from being interfered with from PMAC programs or firmware during the test.

Arguments

dwDevice	Device number.
msgp	Pointer to a user supplied DPRTESTMSGPROC procedure used for message reporting.
Prgp	Pointer to a user supplied DPRTESTPROGRESS procedure used for progress reporting.

Return Value
The number of errors encountered.
See Also
`PmacAbortDPRTest(), DPRTESTMSGPROC , DPRTESTPROGRESS.`

PmacSERGetPort()

```
DWORD PmacSERGetPort(DWORD dwDevice); ets the current serial port number for the device.
```

Arguments

dwDevice	Device number.

Return Value
Port number. 1 = COM1 etc.

ASCII COMMUNICATION FUNCTIONS

GetUSResponse(), USReadReady()

```
BOOL CALLBACK GetUSResponse(DWORD dwDevice, PCHAR response, UINT maxchar)
BOOL CALLBACK USReadReady(DWORD dwDevice)
```

If the code in PMAC has run time errors (always caused by an illegal or badly timed "CMD" statement) PMAC will send an error code string out the active ASCII port. Also if the code in PMAC has any "SEND" commands there will be ASCII responses in PMAC 's output queue. If several of these happen and the host PC is not checking for these unsolicited responses, problems can occur. Typically, this issue is prevented by regularly calling one of the following functions:
PmacFlush()
PmacGetBufferA()
PmacGetControlResponseX()
PmacGetLine()
PmacGetResponseX()

These routines check for unsolicited responses. If one is found it is buffered within the driver. To check if any unsolicited responses have been queued up, call USReadReady(). USReadReady() returns true if there is a response pending. To get a pending unsolicited response string, call GetUSResponse().

Arguments

dwDevice	Device number.
response	Pointer to string buffer. PMAC's response is placed there by function.
maxchar	Maximum characters to copy into response parameter

Return Value
True if successful.

PmacFlush()

```
void PmacFlush(DWORD dwDevice);
```

PmacFlush() is meant to cancel any previously sent commands and clear out both of PMACs ASCII communication queues (input and output queues; both 256 characters in size). This can be useful when:

A) Matching an ASCII string sent to PMACs input ASCII queue, with the corresponding PMAC response put in the output ASCII queue.

B) Telling PMAC to cancel processing a large query from the host PC. For example, if a huge memory dump is requested, it could take a long time for PMAC to send this information out to the host. If for some reason the host wished to abort mid-stream, it could do so with a call to PmacFlush().

This is especially useful if you are not sure what PMAC has been doing before, because it makes sure nothing else comes through before the expected response.

Arguments

dwDevice	Device number.

Return Value
None.

PmacGetAsciiComm()

```
enum ASCIIMODE PmacGetAsciiComm(DWORD dwDevice);
```

Returns the current communications mode. (BUS or DPRAM). Use PmacSetAsciiComm() to set this value and switch between BUS and DPRAM ASCII communication. There is negligible gain when switching from BUS to DPRAM ASCII communication, since, most of the communication time is waiting for PMAC to respond, and not the actual transfer of character data to and from PMACs internal ASCII queues. Most use BUS communication as a means to free up additional DPRAM space.

Arguments

dwDevice	Device number.

Return Value
ASCIIMODE enumeration.

0 = BUS

1 = DPRAM

PmacGetBufferA()

```
long PmacBUSGetBufferA(DWORD dwDevice,PCHAR response,UINT maxchar);
```

PmacGetBufferA() queries PMAC for a full multi-line response. Program listings, or memory dumps are examples of multi-line responses. This function will wait for one of the following conditions to occur before returning:

1. A *timeout* period.
2. PMAC has the full response (as determined by the ACKnowledge character (ASCII 6) received at the end of the response).
3. The maximum number of characters, maxchar, was received.

If PmacGetBufferA() does not succeed on its first attempt to retrieve the response it may be that the PMAC device has no response available or a Timeout has occurred waiting for PMAC to respond.

Arguments

dwDevice	Device number.
response	Pointer to string buffer. PMAC's response is placed there by function.
maxchar	Maximum characters to copy into response parameter

Return Value

The number of characters received.

PmacGetControlResponseA(), PmacGetControlResponseExA()

```
* long PmacGetControlResponseA(DWORD dwDevice,PCHAR response,UINT maxchar, CHAR ctl_char)
long PmacGetControlResponseExA(DWORD dwDevice,PCHAR response,UINT maxchar, CHAR ctl_char)
```

* = This routine is considered obsolete, however, it has been left in for backward compatibility.

PmacGetControlResponse*ExA*() sends a control character to PMAC and potentially returns the ASCII response from PMAC, similar to PmacGetResponse*ExA*().

Arguments

dwDevice	Device number.
response	Pointer to string buffer. PMAC's response is placed there by function.
maxchar	Maximum characters to copy into response parameter
ctl_char	Control character to send to PMAC.

Return Value

PmacGetControlResponse**A**(): Number of characters returned in response parameter.
PmacGetControlResponse**ExA**(): The most significant byte contains the status code, and all lower bytes contain the number of characters returned in response parameter. See "Error Handling - ASCII Communication" for a detailed explanation.

PmacGetLineA(), PmacGetLineExA()

```
* long PmacGetLineA(DWORD dwDevice, PCHAR response, UINT maxchar);
long PmacGetLineExA(DWORD dwDevice, PCHAR response, UINT maxchar);
```

* = This routine is considered obsolete, however, it has been left in for backward compatibility.

PmacGetLineA() queries PMAC for a line response. This function will wait for one of the following conditions to occur before returning:

- A *timeout* period.
- PMAC has a response (A Carraige Return (ASCII 13) or an ACKnowledge character (ASCII 6) was received from PMAC).
- The maximum number of characters, maxchar, was received.

If PmacGetLineA() does not succeed on its first attempt to retrieve the response it may be that the PMAC device has no response available or a Timeout has occurred waiting for PMAC to respond. You can use the PmacReadReady() function to first determine if a response is available.
If you are considering using this function, you will want to make your code thread safe by using both LockPmac() and ReleasePmac(). See "Thread-Safe ASCII Communications" for more information.

Arguments

dwDevice	Device number.
response	Pointer to character buffer to receive character string from PMAC.
maxchar	Maximum number of characters to be received from PMAC.

Return Value
The upper byte contains the status of the call, whereas all lower bytes contain the number of characters received from PMAC. If no characters were received from PMAC, check the upper bytes status code for a potential error code. See "Error Handling - ASCII Communication" for a detailed explanation.

Return Value
Number of characters received (including handshake characters).
FALSE (0) One of the following occurred:

1. No characters in PMAC's buffer queue
2. An error occurred
3. Communication not established

Error Codes

This routine initially sets the internal error flag to FALSE (0). Therefore calling PmacGetError() after this function will inform of any errors that occurred.

PmacGetResponseA(),PmacGetResponseExA()

```
* long PmacGetResponseA(DWORD dwDevice,PCHAR response,UINT maxchar,PCHAR command);
long PmacGetResponseExA(DWORD dwDevice,PCHAR response,UINT maxchar,PCHAR command);
```

* = This routine is considered obsolete, however, it has been left in for backward compatibility.

Most if not all of your communication with the PMAC can be handled with the PmacGetResponseA() function. This function will send a command string (i.e. "#1j+, "?", "Open Prog1", etc.) to the PMAC and retrieve and place any pending responses within a response buffer for your use. This is an efficient and *safe* function to use. The word *safe* is emphasized because there are functions (i.e. PmacSendCharA(), PmacGetLineA() and PmacSendLineA()) which, if not used carefully can cause un-synchronized communication (the mismatching of PMAC Commands to PMAC responses). PmacGetResponseExA () always matches the command string with the response string or else it "times out."

Arguments

dwDevice	Device number.
response	Pointer to string buffer to copy the PMAC's response into.
maxchar	Maximum characters to copy.
command	Pointer to NULL terminated string to be sent to the PMAC as a question/command.

Return Value
The upper byte contains the status of the call, whereas all lower bytes contain the number of characters received from PMAC. If no characters were received from PMAC, check the upper bytes status code for a potential error code. See "Error Handling - ASCII Communication" for a detailed explanation.

Return Value
If successful, this function returns the number of characters received, including handshake characters. Otherwise FALSE (0) which implies of course that an error occurred, or no characters were received since PMAC was not required to respond.
Error Codes
This routine initially sets the internal error flag to FALSE (0). Therefore, calling PmacGetError() after this function will inform of any errors that occurred.

PmacReadReady()

```
BOOL PmacReadReady(DWORD dwDevice);
```

This function determines if PMAC has a response waiting on the current communications channel (BUS, DPRAM, SERIAL). Use this function before using other query functions (PmacGetLineA() or PmacGetBufferA*()* for example) to prevent timeout errors, thus speeding communication throughput.
Arguments

dwDevice	Device number.

Return Value
TRUE if PMAC has a response waiting.

PmacSendCharA()

```
BOOL PmacSendCharA(DWORD dwDevice,CHAR outchar);
```

Sends a single character to the PMAC. Does not append EOL (Carraige Return for PMAC).
Arguments

dwDevice	Device number.
outchar	Character sent to PMAC.

Return Value
TRUE if success.

PmacSendCommandA()

```
void PmacSendCommandA(DWORD dwDevice,PCHAR command);
```

Sends a string buffer to PMAC and flushes out any response from PMAC. Also if PMAC is responding to any previous communication exchange, it is cleared.

Note:

This function is robust but slow since it will time out at least once to assure there is nothing left in PMAC's output ASCII queue. **It's almost always better to use GetResponse() instead.**

Arguments

dwDevice	Device number.
command	Pointer to NULL terminated string sent to PMAC.

Return Value

None.

PmacSendLineA()

```
long PmacSendLineA(DWORD dwDevice,PCHAR command);
```

Sends a string buffer to PMAC and disregards any response from PMAC.

Arguments

dwDevice	Device number.
command	Pointer to NULL terminated string sent to PMAC.

Return Value

Number of characters sent.

PmacSetAsciiComm()

```
BOOL PmacSetAsciiComm(DWORD dwDevice,enum ASCIIMODE m);
```

Used to switch select either BUS or DPRAM ASCII communication. There is negligible gain when switching from BUS to DPRAM ASCII communication, since, most of the communication time is waiting for PMAC to respond, and not the actual transfer of character data to and from PMACs internal ASCII queues. Most use BUS communication as a means to free up additional DPRAM space. Therefore, this function is rarely used.

Note:

This will not change the communication method used when PComm32PRO starts upon initialization. To do this see PmacConfigure().

Arguments

dwDevice	Device number.
m	Mode: 0 = BUS, 1 = DPRAM

Return Value

TRUE if success.

See Also

PmacConfigure()

DOWNLOADING FUNCTIONS

PmacAbortDownload()

```
void PmacAbortDownload(DWORD dwDevice);
```

Calling this function will cause a download in progress to be aborted. This applies for driver downloading functions PmacDownload() and PmacDownloadFirmwareFile().

Arguments

dwDevice	Device number.

Return Value

None.

See Also

PmacDownload(), and PmacDownloadFirmwareFile()

PmacAddDownloadFileA()

```
long PmacAddDownloadFileA(DWORD dwDevice,PCHAR inifile,PCHAR szUserId,PCHAR szDLFile);
```

This utility function was added to assist in the use of PmacMultiDownload() function. PmacMultiDownload() downloads each file listed within a text file, referred to as the list file. PmacAddDownloadFileX appends a specified file name to the list file, *inifile*. If the file to be added to the list already in the list, nothing is done. To remove a file from the list use PmacRemoveDownloadFileA().

All parameters to this function are case insensitive and white space characters will be ignored.

Arguments

inifile	Initialization file to add new download file to. If NULL, the default is "WIN.INI".
szUserId	User id string, this associates the file being added to a given application or user.
szDLFIle	Name of the file to add to download list.

Return Value

Greater than zero(0) if successful, else zero(0);

See Also

PmacMultiDownload(), PmacRemoveDownloadFileA()

PmacDownloadA()

```
long  PmacDownloadA(DWORD dwDevice, DOWNLOADMSGPROC msgp, DOWNLOADGETPROC getp,
     DOWNLOADPROGRESS pprg, PCHAR filename, BOOL macro, BOOL map, BOOL log, BOOL dnld);
```

This function takes an ASCII file, processes it, and downloads it from the PC to the PMAC. Processing includes MACRO parsing and compiling PLCs, for example. This function can generate several residual files, as described in the table below.

File name	Usage
Filename.EXT	***Original file with the original EXTension (should not be *.PMA, *.56K, *.LOG, *.MAP).***
Filename.PMA	After parsing the file for #define, #includes and other MACRO's this file is generated. It may be downloaded if no compiling is necessary.
Filename.56K	This file will be created if the Filename.PMA was compiled. Compilation occurs when the *macro* parameter is set to TRUE.
Filename.LOG	The status of the download at each stage is recorded when the *log* parameter is set to TRUE.
Filename.MAP	A lookup table is created when MACRO definitions exist. They are recorded and saved to a file when the *map* parameter is set to TRUE.

Arguments

dwDevice	Device number.
msgp	Pointer to message procedure pointer. (DOWNLOADMSGPROC) If NULL no function is called.
getp	Pointer to line retrieval function. (DOWNLOADGETPROC) If NULL no function is called.
pprg	Pointer to download progress function. (DOWNLOADPROGRESS) If NULL no function is called.
filename	Path of file to download.
macro	Flag to parse for macros.
map	Flag to create a map file created from macros.
log	Flag to create a log file. This is the same messages as sent to the "msgp" procedure.
dnld	Flag indicating to send final parsed file to the PMAC.

Return Value

Non-zero if successful. 0 if a failure occurs.

See Also

DOWNLOADMSGPROC, DOWNLOADPROGRESS, (), "Downloading To PMAC" in Users Manual.

PmacDownloadFirmwareFile()

```
long  PmacDownloadFirmwareFile(DWORD dwDevice, DOWNLOADMSGPROC msgp, DOWNLOADPROGRESS prgp,
    PCHAR filename);
```

Flash ROM installation function. PMAC must be in BOOTSTRAP mode in order to download new firmware. Usually a jumper (E3 on PMAC2, E51 on PMAC) puts PMAC in a BOOTSTRAP mode. If talking serially the baud rate must be set to 38400.

Arguments

dwDevice	Device number.
Msgp	Pointer to message procedure pointer. (DOWNLOADMSGPROC) If NULL no function is called.
prgp	Pointer to download progress function. (DOWNLOADPROGRESS) If NULL no function is called.
filename	Path of firmware binary file to download.

Return Value

The error count. Zero(0) if no errors.

PmacMultiDownload()

```
long PmacMultiDownload(DWORD dwDevice, DOWNLOADMSGPROC msgp,  PCHAR outfile, PCHAR inifile,
    PCHAR szUserId, BOOL macro, BOOL map, BOOL log, BOOL dnld);
```

The use of this function is **required** in those systems in which:

A) Have files with PLCC's within them, all meant to be within a single PMAC. Or

B) Have more than one application that will be downloading PLCC's to the PMAC.

The critical issue here is that all PLCC's must be downloaded at the same time for a given PMAC. This function will resolve the Multi-Application problem associated with PMAC compiled PLCs. Similar to the PmacDownload() routine, a file is downloaded to PMAC. The difference here however is that instead of specifying one file, many are specified. The names of files to be downloaded are stored in a 'list file'. The 'list file', if not specified, will default to the WIN.INI file.

For example the syntax within the WIN.INI file may look like this:

```
[PMACDOWNLOAD]
DOWNLOADFILE1="WINDEMO C:\PCOMM\PCOMM\TESTPLCC.PLC"
DOWNLOADFILE2="WINDEMO C:\PCOMM\PCOMM\NCPANEL.PLC"
DOWNLOADFILE3="WINDEMO C:\PCOMM\WIZDOM\SPINDLE.PLC"
DOWNLOADFILE4="PEWIN C:\PCOMM\WIZDOM\SPINDLE.PLC"
DOWNLOADFILE5="USERPROGRAM C:\USER3\BIFF.PLC"
```

Caveat: All PMAC files that include other files need to use the fully qualified path. For example, if one of the files to be downloaded is myfile.plc, and within myfile.plc there is an include statement for myfile.h, the include statement must use the fully qualified path (i.e. #include "c:\mydirect\myfile.h")

This function has added flexibility with the use of the szUserId parameter (described in the table below). MultiDownload() makes a temporary file, by the name of **outfile**, and places an include statement for every file within the initialization file, **iniFile**. It then calls the PmacDownload() function. The **outfile** parameter will also be the name of any intermediary files created during the download, such as the log, map, and, compiled files (i.e. files produced will be **outfile**.log **outfile**.map etc.).

Arguments

outfile	Specifies the name of the output file. This parameter also specifies the path of the files generated by this function. If NULL, the default is "PMACDNLD", path of file is active directory. It's best to supply a path for consistency. This also indirectly specifies the name of the log, map and other intermediary files that will be created.
IniFile	Initialization file to be used. If Null the defaultis "WIN.INI".
szUserId	Application user id string. If this is NULL all within the iniFile are downloaded within "PMACDOWNLOAD" section. Otherwise, only those preceded by the "user id" string within the iniFile will be downloaded.
macro	Flag to parse for macros.
map	Flag to create a map file created from macros.
log	Flag to create a log file. This is the messages sent to “msgp”.
dnld	Flag indicating to send final parsed file to PMAC.

Return Value
TRUE, if all went well, otherwise -1 * (Number of errors) or FALSE if memory or file could not be allocated/opened, or if WIN.INI did not contain files to be downloaded.
See Also
PmacAddDownloadFileA(), PmacRemoveDownloadFileA(), "Downloading To PMAC" in the Users Manual.

PmacRemoveDownloadFile()

```
long PmacRemoveDownloadFile(DWORD dwDevice, PCHAR inifile, PCHAR szUserId, PCHAR szDLFile);
```

This utility function was added to assist in the use of the PmacMultiDownloadA() function. It removes a specified file name from the list file. It also may remove all files associated with a particular id string, by setting the *szDLFile* parameter to NULL.
All parameters to this function are case insensitive and white space characters will be ignored. To add a file from the list use PmacAddDownloadFileA().
Arguments

inifile	Initialization file to remove file from. If NULL, the default is "WIN.INI"
szUserId	User id string, this associates the file being added to a given application or user.
SzDLFile	Name of the file to remove from download list. If Null, it will remove all files with the szUserId tag.

Return Value
TRUE if file located and removed, else FALSE.
See Also
PmacAddDownloadFileA(), PmacRemoveDownloadFileA(), "Downloading To PMAC" in the Users Manual.

PmacSetMaxDownloadErrors()

```
void PmacSetMaxDownloadErrors(UINT max);
```

Sets the limit of errors allowed during a download before the function aborts automatically. Valid for use with functions, PmacDownload(), and PmacMultiDownload().

Arguments

max	Number of errors allowed before download will terminate

Return Value

None.

See Also

PmacDownload(), PmacMultiDownload(), and "Downloading To PMAC" in the Users Manual.

DUAL PORTED RAM CONTROL PANEL

PmacDPRControlPanel()

```
BOOL PmacDPRControlPanel(DWORD dwDevice, long onoff);
```

This function is used to enable or disable the PMAC DPR Control Panel feature.

Arguments

dwDevice	Device number.
onoff	To turn on use TRUE (1) else FALSE (0)

Return Value

This function returns the active status of the DPR Control Panel Feature. TRUE(1) is on, FALSE (0) is off.

See Also

"Using the DPR Control Panel" in the Users Manual, PmacDPRGet*Action*Bit() and PmacDPRSet*Action*Bit().

*PmacDPRGet*Action*Bit() and PmacDPRSet*Action*Bit()*

```
void PmacDPRSetJogPosBit(DWORD dwDevice, long motor, long onoff);
long PmacDPRGetJogPosBit(DWORD dwDevice, long motor);
void PmacDPRSetJogNegBit(DWORD dwDevice, long motor, long onoff);
long PmacDPRGetJogNegBit(DWORD dwDevice, long motor);
void PmacDPRSetJogReturnBit(DWORD dwDevice,long motor,long onoff);
long PmacDPRGetJogReturnBit(DWORD dwDevice, long motor);
void PmacDPRSetRunBit(DWORD dwDevice, long cs, long onoff);
long PmacDPRGetRunBit(DWORD dwDevice, long cs);
void PmacDPRSetStopBit(DWORD dwDevice, long cs, long onoff);
long PmacDPRGetStopBit(DWORD dwDevice, long cs);
void PmacDPRSetHomeBit(DWORD dwDevice, long cs, long onoff);
long PmacDPRGetHomeBit(DWORD dwDevice, long cs);
void PmacDPRSetHoldBit(DWORD dwDevice, long cs, long onoff);
long PmacDPRGetHoldBit(DWORD dwDevice, long cs);
long PmacDPRGetStepBit(DWORD dwDevice, long cs);
void PmacDPRSetStepBit(DWORD dwDevice, long cs, long onoff);
```

These functions are used to control the PMAC DPR Control Panel feature. The PmacDPRSet*Action*Bit() and PmacDPRGet*Action*Bit() routines set/get a bit in DPR which indicate to PMAC to perform the action on the designated motor/coordinate system.

Arguments

dwDevice	Device number.
onoff	To turn on use TRUE (1) else FALSE (0)
motor	Motor number (1-8)
cs	Coordinate System Number (1-8)

Return Value

The PmacDPRGet*Action*Bit() routines return the state of the bit (0/1 for disabled/enabled). The PmacDPRSet*Action*Bit() functions return nothing.

See Also

"Using the DPR Control Panel" in the Users Manual, PmacDPRControlPanel(), PmacDPRGetRequestBit() and PmacDPRSetRequestBit()

PmacDPRGetFOEnableBit() and PmacDPRSetFOEnableBit()

```
long PmacDPRGetFOEnableBit(DWORD dwDevice, long mtr_crd);
void PmacDPRSetFOEnableBit(DWORD dwDevice, long mtr_crd, long on_off);
```

These functions are used to control the PMAC DPR Control Panel **F**eedrate **O**verride feature. The PmacDPRSetFOEnableBit() and PmacDPRGetFOEnableBit() routines set/get a bit in DPR which tell PMAC whether or not to use the feedrate specified in the DPR. From PMAC’s perspective, the enable bit is read only. The value of the feedrate is set with PmacDPRSetFOValue().

Arguments

dwDevice	Device number.
mtr_crd	Motor or Coordinate System Number (1-8)
onoff	To turn on use TRUE (1) else FALSE (0)

Return Value

The PmacDPRGetRequestBit() routines return the state of the bit (0/1). The PmacDPRSetRequestBit() functions return nothing.

See Also

"Using the DPR Control Panel" in the Users Manual, PmacDPRControlPanel(), PmacDPRSetFOValue()

PmacDPRGetFOValue() and PmacDPRSetFOValue()

```
void PmacDPRSetFOValue(DWORD dwDevice, long cs, long value);
long PmacDPRGetFOValue(DWORD dwDevice, long cs);
```

These functions are used to control the PMAC DPR Control Panel **F**eedrate **O**verride feature. The PmacDPRSetFOValue() and PmacDPRGetFOValue() routines set/get the feedrate override value in DPR. Whether or not the feedrate is used depends on the arguments used in the last call to PmacDPRSetFOEnableBit().

The *value* parameter for PmacDPRSetFOValue() may be anything from 1 to 32,767 in units of 1/32,768 msec. The *value* represents the "elapsed time" PMAC uses in its trajectory update equations each servo cycle. If it matches the true time, the trajectories will go at the programmed speeds. If it is greater than the true time, the trajectories will go faster, if it is less, they will go slower. V*alue* corresponds to values of 0 to 8,388,352 in units of I10 (1/8,388,608 msec). At the default I10 value of 3,713,707, this corresponds to a feedrate override (%) values from 0 to 225.87; for real-time execution (%100) *value* should be 14507.

Arguments

dwDevice	Device number.
cs	Coordinate System Number (1-8)
value	See explanation above

Return Value

The PmacDPRGetRequestBit() routines return the state of the bit (0/1). The PmacDPRSetRequestBit() functions return nothing.

See Also

"Using the DPR Control Panel" in the Users Manual, PmacDPRControlPanel(), PmacDPRSetFOValue()

PmacDPRGetRequestBit() and PmacDPRSetRequestBit()

```
long PmacDPRGetRequestBit(DWORD dwDevice, long mtr_crd);
void PmacDPRSetRequestBit(DWORD dwDevice,long mtr_crd,long onoff);
```

These functions are used to control the PMAC DPR Control Panel feature. The PmacDPRSetRequestBit() and PmacDPRGetRequestBit() routines set/get a bit in DPR which indicate to PMAC to look at all the *action* bits that have been set and act on them. PMAC will reset the request bit once the requested actions have been processed. The request bit can therefore be considered a handshaking bit between the host computer and the PMAC. Host sets, PMAC notes the set bit, PMAC performs requested actions, PMAC resets the request bit.

Arguments

dwDevice	Device number.
mtr_crd	Motor or Coordinate System Number (1-8)
onoff	To turn on use TRUE (1) else FALSE (0)

Return Value

The PmacDPRGetRequestBit() routines return the state of the bit (0/1). The PmacDPRSetRequestBit() functions return nothing.

See Also

"Using the DPR Control Panel" in the Users Manual, PmacDPRControlPanel().

DPR REAL TIME FIXED DATA BUFFER

DPR Real Time Fixed Data Buffer Initialization and Handshaking

Startup/Shutdown

```
BOOL PmacDPRRealTimeEx(DWORD dwDevice, long mask, UINT period, int on
)
void PmacDPRSetRealTimeMotorMask ( DWORD dwDevice, long mask )
```

HandShaking

```
BOOL PmacDPRUpdateRealTime( DWORD dwDevice )
```

Note: The only significant difference between the Turbo and non-Turbo implementation, is the interpretation of the **mask** parameter of PmacDPRRealTimeEx(). Other than that, it is only the Data query methods that differ. Turbo’s Realtime data is motor only specific, whereas the non-Turbo contains more generic information.

The PmacDPRRealTimeEx() function enables/disables the DPR real time fixed data buffer. When enabled, all the data query functions above may be used. The **period** parameter in PmacDPRRealTimeEx() specifies how often in servo-cycles PMAC will update the data in this buffer. The data for motor numbers 1- **`mask`** will be updated in DPR where **`mask`** is the second parameter in PmacDPRRealTimeEx () function. The final parameter **on** will enable this feature if set to 1, otherwise it will disable it. Once enabled the range of motors updated can be modified via the PmacDPRSetRealTimeMotorMask() routine.
The data is refreshed within the DPR when the PmacDPRUpdateRealTime() function is called. Call this before querying data.
Note the correct sequence for handshaking:

1. Refresh the data within the DPR via the DPRDoRealTimeHandshake() routine
2. Call the Data Query functions to read the DPR Real Time Data buffer.

Arguments

dwDevice	Device number.
period	In units of servo periods (See I10 in PMAC Users Manual)
on_off	Turn on (1) or off (0)
mask	1-mask motors will be updated.

DPR Real Time Fixed Data Buffer Query Routines

Global

```
long  PmacDPRGetServoTimer(DWORD dwDevice);
```

The return value of this routine reflects PMAC's servo counter register located at PMAC address 0.

```
BOOL PmacDPRSysServoError(DWORD dwDevice);
```

The return value of this function indicates if PMAC could not properly complete its servo routines. This is a serious error condition. It returns FALSE if the servo operations have been completing properly.

```
BOOL PmacDPRSysReEntryError(DWORD dwDevice);
```

Returns TRUE if a real-time interrupt task has taken long enough so that it was still executing when the next real-time interrupt came (I8+1 servo cycles later). It stays TRUE until the card is reset or the bit in the global status register is reset manually.

```
BOOL PmacDPRSysMemChecksumError(DWORD dwDevice);
```

The return value is TRUE if a checksum error has been detected for either the PMAC firmware or the user program buffer space. PmacDPRSysPromChecksumError() distinguishes between the two cases.

```
BOOL PmacDPRSysPromChecksumError(DWORD dwDevice);
```

Returns TRUE if a firmware checksum error has been detected in PMAC's memory. The return value is FALSE if a user program checksum has been detected, or if no memory checksum error has been detected. PmacDPRSysMemChecksumError() distinguishes between the two cases.

```
GLOBALSTATUS PmacDPRGetGlobalStatus(DWORD dwDevice);
```

Returns a global status structure, GLOBALSTATUS, defined in the section "Data Types, Structures, Callbacks, and Constants.

Coordinate System

```
BOOL PmacDPRMotionBufOpen(DWORD dwDevice);
```

The return value indicates whether or not a motion program buffer is currently open.

```
BOOL PmacDPRRotBufOpen(DWORD dwDevice);
```

The return value indicates whether or not a rotary buffer is currently open.

```
double PmacDPRGetFeedRateMode(DWORD dwDevice,long coordinate_system, BOOL *mode);
```

NOTE: The “coordinate_system” parameter is a 0 based index. For example, 0 is coordinate system 1, 1 is coordinate system 2 etc.

The return value of this function is TRUE (non-zero) if programmed moves in this coordinate system are currently specified by time (TM or TA), and the move speed is derived. It is FALSE (0) if programmed moves in this coordinate system are currently specified by feedrate (speed; F) and the move time is derived.

Motor

The “`motor`” parameter in these functions, is a 0 based index. For example, 0 is motor 1, 1 is motor 2 etc.

```
SERVOSTATUS PmacDPRMotorServoStatus(DWORD dwDevice,long motor);
```

Returns a global status structure, SERVOSTATUS, defined in the section "Data Types, Structures, Callbacks, and Constants."

```
BOOL CALLBACK PmacDPRDataBlock(DWORD dwDevice,long motor);
```

This function returns TRUE when move execution has been aborted because the data for the next move section was not ready in time. This is due to insufficient calculation time. It is FALSE otherwise.

```
BOOL PmacDPRPhasedMotor(DWORD dwDevice,long motor);
```

This function returns TRUE when Ix01 is 1 and this motor is being commutated by PMAC; it is FALSE when Ix01 is 0 and this motor is not being commutated by PMAC.

```
BOOL PmacDPRMotorEnabled(DWORD dwDevice,long motor);
```

This function returns TRUE when Ix00 is 1 and the motor calculations are active. It is 0 when Ix00 is 0 and motor calculations are deactivated.

```
BOOL PmacDPRHandwheelEnabled(DWORD dwDevice,long motor);
```

This function returns TRUE when Ix06 is 1 and position following for this axis is enabled. It returns FALSE when IX06 is 0 and position following is disabled.

```
BOOL PmacDPROpenLoop(DWORD dwDevice,long motor);
```

When this function returns TRUE the servo loop for the motor is open, either with outputs enabled or disabled (killed). It returns FALSE when the servo loop is closed (under position control, always with outputs enabled).

```
BOOL PmacDPROnNegativeLimit(DWORD dwDevice,long motor);
```

Returns TRUE when motor actual position is less than the software negative position limit (Ix14), or when the hardware limit on this end (+LIMn – note!) has tripped; it is FALSE otherwise.

```
BOOL PmacDPROnPositiveLimit(DWORD dwDevice,long motor);
```

Returns TRUE when motor actual position is greater than the software positive position limit (Ix13), or when the hardware limit on this end (-LIMn – note!) has tripped; it is FALSE otherwise.

```
void  PmacDPRSetJogReturn(DWORD dwDevice,long motor);
```

This can be used to set the Jog Return Position for the motor specified. The current actual position is used to assign the Jog Return Position.

```
MOTION PmacDPRGetMotorMotion(DWORD dwDevice,long motor);
```

Returns an enumeration based on the motion state of the specified motor.

```
typedef enum { inpos,jog,running,homing,handle,openloop,disabled } MOTION;
double PmacDPRGetCommandedPos(DWORD dwDevice,long motor, double units);
```

This function returns the commanded position of the specified motor. Units are in encoder counts unless the parameter *units* is not one.

```
double PmacDPRPosition(DWORD dwDevice,long motor,double units);
```

This function returns the actual position of the specified motor in units of encoder counts provided the *units* variable is unity.

```
double PmacDPRFollowError(DWORD dwDevice,long motor,double units);
```

This function returns the following error of the specified motor in units of encoder counts provided the *units* parameter is unity.

```
double PmacDPRGetVel(DWORD dwDevice,long motor,double units);
```

This function returns the velocity of the specified motor in units of (1/(Ix09*32) counts per servo cycle.

```
void PmacDPRGetMasterPos(DWORD dwDevice,long motor,double units,double *the_double);
```

This function returns the master position of the specified motor in units of encoder counts unless the *units* parameter is unity.

```
void PmacDPRGetCompensationPos(DWORD dwDevice,long motor,double units,double *the_double);
```

This function returns the compensation position of the specified motor in units of encoder counts unless the *units* parameter is unity.

```
DWORD PmacDPRGetPrevDAC(DWORD dwDevice,long motor);
```

This function returns the DAC value of the previous servo cycle. It is in units of 1/256 DAC bits.

```
DWORD PmacDPRGetMoveTime(DWORD dwDevice,long motor);
```

This function returns the time (in milliseconds) left in the currently executing move.

DPR REAL TIME FIXED DATA BUFFER (TURBO)

DPR Real Time Fixed Data Buffer Initialization (Turbo)

Startup/ShutDown

```
BOOL PmacDPRRealTimeEx(DWORD dwDevice, long mask, UINT period, int on
)
void PmacDPRSetRealTimeMotorMask( DWORD dwDevice, long mask )
```

HandShaking

```
BOOL PmacDPRUpdateRealTime( DWORD dwDevice )
```

Note: The only significant difference between the Turbo and non-Turbo implementation, is the interpretation of the **mask** parameter of PmacDPRRealTimeEx(). Other than that, it is only the Data query methods that differ. Turbo's Realtime data is motor only specific, whereas the non-Turbo contains more generic information.

The PmacDPRRealTimeEx() function enables/disables the DPR real time fixed data buffer. The final parameter **on** will enable this feature if set to 1, otherwise it will disable it. When enabled, all the data query functions above may be used. The **period** parameter in PmacDPRRealTimeEx() specifies how often in servo-cycles PMAC will update the data in this buffer. The **mask** parameter is specifies which of the possible 32 motor data sets to update. Bit 0, the least significant bit, enables or disables the first motor by setting it to 1 or 0. Bit 1 enables or disables the second motor etc… The PmacDPRSetRealTimeMotorMask() routine can also be used to modify which motor data sets are being updated after initialization has been done.
Typical usage of these routines is shown below and also is available in the supplied example source code (PmacTest application).

Arguments

dwDevice	Device number.
period	In units of servo periods (See I10 in PMAC Users Manual)
on_off	Turn on (1) or off (0)
mask	Used to specify what motor data to update in the buffer. Bit 0 = motor 1, Bit 1 = motor 2 etc. 1=report, 0 = do not report.

DPR Real Time Fixed Data Buffer Query Routines (Turbo)

Global

```
long  PmacDPRGetServoTimer(DWORD dwDevice);
```

The return value of this routine reflects PMAC's servo counter register located at PMAC address 0.

Motor

The “`motor`” parameter in these functions, is a 0 based index. For example, 0 is motor 1, 1 is motor 2 etc.

```
SERVOSTATUS PmacDPRMotorServoStatus(DWORD dwDevice,long motor);
```

Returns a global status structure, SERVOSTATUS, defined in the section "Data Types, Structures, Callbacks, and Constants".

```
BOOL CALLBACK PmacDPRDataBlock(DWORD dwDevice,long motor);
```

This function returns TRUE when move execution has been aborted because the data for the next move section was not ready in time. This is due to insufficient calculation time. It is FALSE otherwise.

```
BOOL PmacDPRPhasedMotor(DWORD dwDevice,long motor);
```

This function returns TRUE when Ix01 is 1 and this motor is being commutated by PMAC, it is FALSE when Ix01 is 0 and this motor is not being commutated by PMAC.

```
BOOL PmacDPRMotorEnabled(DWORD dwDevice,long motor);
```

This function returns TRUE when Ix00 is 1 and the motor calculations are active. It is 0 when Ix00 is 0 and motor calculations are deactivated.

```
BOOL PmacDPRHandwheelEnabled(DWORD dwDevice,long motor);
```

This function returns TRUE when Ix06 is 1 and position following for this axis is enabled. It returns FALSE when IX06 is 0 and position following is disabled.

```
BOOL PmacDPROpenLoop(DWORD dwDevice,long motor);
```

This function returns TRUE when the servo loop for the motor is open, either with outputs enabled or disabled (killed). It returns FALSE when the servo loop is closed (under position control, always with outputs enabled).

```
BOOL PmacDPROnNegativeLimit(DWORD dwDevice,long motor);
```

This function returns TRUE when motor actual position is less than the software negative position limit (Ix14), or when the hardware limit on this end (+LIMn – note!) has tripped; it is FALSE otherwise.

```
BOOL PmacDPROnPositiveLimit(DWORD dwDevice,long motor);
```

This function returns TRUE when motor actual position is greater than the software positive position limit (Ix13), or when the hardware limit on this end (-LIMn – note!) has tripped; it is FALSE otherwise.

```
void  PmacDPRSetJogReturn(DWORD dwDevice,long motor);
```

This can be used to set the Jog Return Position for the motor specified. The current actual position is used to assign the Jog Return Position.

```
MOTION PmacDPRGetMotorMotion(DWORD dwDevice,long motor);
```

Returns an enumeration based on the motion state of the specified motor.

```
typedef enum { inpos,jog,running,homing,handle,openloop,disabled } MOTION;
double PmacDPRGetCommandedPos(DWORD dwDevice,long motor, double units);
```

This function returns the commanded position of the specified motor. Units are in encoder counts unless the parameter *units* is not one.

```
double PmacDPRPosition(DWORD dwDevice,long i,double units);
```

This function returns the actual position of the specified motor in units of encoder counts provided the *units* variable is unity.

```
double PmacDPRFollowError(DWORD dwDevice,long motor,double units);
```

This function returns the following error of the specified motor in units of encoder counts provided the *units* parameter is unity.

```
double PmacDPRGetVel(DWORD dwDevice,long motor,double units);
```

This function returns the velocity of the specified motor in units of (1/(Ix09*32) counts per servo cycle.

```
void PmacDPRGetMasterPos(DWORD dwDevice,long motor,double units,double *the_double);
```

This function returns the master position of the specified motor in units of encoder counts unless the *units* parameter is unity.

```
void PmacDPRGetCompensationPos(DWORD dwDevice,long motor,double units,double *the_double);
```

This function returns the compensation position of the specified motor in units of encoder counts unless the *units* parameter is unity.

```
DWORD PmacDPRGetPrevDAC(DWORD dwDevice,long motor);
```

This function returns the DAC value of the previous servo cycle. It is in units of 1/256 DAC bits.

DPR BACKGROUND FIXED DATA BUFFER

DPR Background Fixed Data Buffer Initialization and Handshaking

```
BOOL PmacDPRBackground(DWORD dwDevice, long on_off);
BOOL PmacDPRSetBackground(DWORD dwDevice);
void PmacDPRSetMotors(DWORD dwDevice,UINT n);
```

The "`n`" parameter in the `PmacDPRSetMotors` () routine is a 1 based index. For example, 1 is motor 1, 2 is motor 2 etc.

To enable this feature, call the PmacDPRBackground() routine with the on_off parameter set to a non-zero value. To retrieve the information, use the provided DPR Background Fixed Data buffer query functions. Information for motors/coordinate systems 1-*n* will be updated where *n* is the second parameter of the PmacDPRSetMotors() function. You may call DPRSetMotors() before or after PmacDPRBackground().
When enabled, all the query functions above may be used to:

1. Query PMAC's control panel, thumbwheel and machine I/O connector status.
2. Get motor target,bias and commanded position registers. Also query a motor's status word.
3. Query a coordinate system's status word, program status, program remaining, time remaining in move and acceleration, and finally, the coordinate system's currently executing line.

Once enabled, call the DPRSetBackground() function to assure that the data query functions get fresh data. All data query functions have this function within it, relieving you from dealing with the handshaking details. A typical sequence of function calls to data would be:

```
PmacDPRSetMotors(p,4);// Update data for motors/cs 1 - 4
p  =  PMAC_DEVICE_NUMBER;
while(!DONE){
        printf("Target Position: %11.1lf\n",PmacDPRGetTargetPos(p, mtrcrd,1.0/(96.0*32.0)));
        printf("Bias Position: %11.1lf\n",PmacDPRGetBiasPos(p,mtrcrd,1.0/(96.0*32.0)));
        aDouble=PmacDPRCommanded(p,mtrcrd,'A');
        printf("Commanded Position A: %11.4lf\n",aDouble);
        aDouble=PmacDPRCommanded(p,mtrcrd,'B');
        printf("PmacCommanded Position B: %11.4lf\n",aDouble);
        aDouble=PmacDPRCommanded(p,mtrcrd,'C');
        printf("Commanded Position C: %11.4lf\n",aDouble);
        .
        .
        printf("WFE:%d\n",PmacDPRWarnFError(p,mtrcrd));
        printf("FFE:%d\n",PmacDPRFatalFError(p,mtrcrd));
        printf("AMP FAULT:%d\n",PmacDPRAmpFault(p,mtrcrd));
        printf("POSITION LIMIT:%d\n",PmacDPROnPositionLimit(p,mtrcrd));
        printf("HOME COMPLETE:%d\n",PmacDPRHomeComplete(p,mtrcrd));
        printf("MOTOR IN POSITION:%d\n",PmacDPRInposition(p,mtrcrd));
}
```

DPR Background Fixed Data Buffer Query Routines

Data Query functions

Motor Specific

The "`motor`" parameter in the functions below is a 0 based index. For example, 0 is motor 1, 1 is motor 2 etc.

```
double PmacDPRCommanded(DWORD dwDevice, long crd, char axchar);
```

This routine returns the commanded position for an axis (described by *axchar*) in coordinate system *crd* in the same units as used to define the axis. The crd parameter is a zero based index therefore 0 is for coordinate system 1, 1 is for coordinate system 2 etc.

```
double PmacDPRGetVel(DWORD dwDevice, long motor, double units);
```

This function returns the velocity of the selected motor in units of counts per minute if the *units* parameter is set to 1.

```
double PmacDPRVectorVelocity(DWORD dwDevice,long num,long motors[], double units[]);
```

This function returns the net (or vector) velocity of the chosen *motors[]* in units of counts per minute if the *units[]* parameter is unity.

```
double PmacDPRGetTargetPos(DWORD dwDevice,long motor,double posscale);
```

The target position of the specified *motor* is returned by this function. The units are in counts unless *posscale* is not unity.

```
double  PmacDPRGetBiasPos(DWORD dwDevice,long motor, double posscale);
```

The bias position register is returned by this function in units of counts if the *posscale* parameter is unity.

```
BOOL PmacDPRAmpEnabled(DWORD dwDevice,long motor);
BOOL PmacDPRWarnFError(DWORD dwDevice,long motor);
BOOL PmacDPRFatalFError(DWORD dwDevice,long motor);
BOOL PmacDPRAmpFault(DWORD dwDevice,long motor);
BOOL PmacDPROnPositionLimit(DWORD dwDevice,long motor);
BOOL PmacDPRHomeComplete(DWORD dwDevice,long motor);
BOOL PmacDPRInposition(DWORD dwDevice,long motor);
```

The functions above return the boolean status of a variety of motor flags.

Coordinate System Specific

The "`cs`" parameter in the functions below is a 0 based index. For example, 0 is coordinate system 1, 1 is coordinate system 2 etc.

```
long PmacDPRPe(DWORD dwDevice,long cs);
```

This function returns the program execution status for a given coordinate system, *cs*. The lower 24 bits are used. The low 24 bits are the same as the second word returned on a "??" command from PMAC.

```
long  PmacDPRProgRemaining(DWORD dwDevice,long cs);
```

This function returns the number of program lines remaining for a specified coordinate system, *cs*. Same as the "PR" command for PMAC.

```
long  PmacDPRTimeRemInMove(DWORD dwDevice,long cs);
```

This function returns the time remaining in the current move for a specified coordinate system, *cs*. The value this function returns is not very intuitive or useful for other than display purposes. The time for a single line in a program may be broken up in to 3 parts, acceleration, steady state, and deceleration times.

```
long  PmacDPRTimeRemInTATS(DWORD dwDevice,long cs);
```

This function returns the time remaining in accel/decl when I13>0.

```
BOOL PmacDPRRotBufFull(DWORD dwDevice,long cs);
BOOL PmacDPRSysInposition(DWORD dwDevice,long cs);
BOOL PmacDPRSysWarnFError(DWORD dwDevice,long cs);
BOOL PmacDPRSysFatalFError(DWORD dwDevice,long cs);
BOOL PmacDPRSysRunTimeError(DWORD dwDevice,long cs);
BOOL PmacDPRSysCircleRadError(DWORD dwDevice,long cs);
BOOL PmacDPRSysAmpFaultError(DWORD dwDevice,long cs);
BOOL PmacDPRProgRunning(DWORD dwDevice,long cs);
BOOL PmacDPRProgStepping(DWORD dwDevice,long cs);
BOOL PmacDPRProgContMotion(DWORD dwDevice,long cs);
BOOL PmacDPRProgContRequest(DWORD dwDevice,long cs);
```

The functions above return the boolean status of a variety of coordinate system flags.

Logical Query Functions

The "`cs`" / "motor" parameter in the functions below is a 0 based index. For example, 0 is coordinate system 1, 1 is coordinate system 2 etc.

```
MOTIONMODE  PmacDPRGetMotionMode(DWORD dwDevice,long cs);
```

The motion mode of a chosen coordinate system is returned by this function. The MOTIONMODE enumeration is shown below:

```
typedef enum { linear,rapid,circw,circcw,spline,pvt } MOTIONMODE;
PROGRAM  PmacDPRGetProgramMode(DWORD dwDevice,long cs);
```

The program mode of a chosen coordinate system is returned by this function. The PROGRAMMODE enumeration is shown below:

```
typedef enum { stop,run,step,hold,joghold,jogstop } PROGRAM;

enum MOTION  PmacDPRGetMotorMotion(DWORD dwDevice,long motor);
```

The motion mode of a chosen motor is returned by this function. The MOTION enumeration is shown below:

```
typedef enum { inpos,jog,running,homing,handle,openloop,disabled } MOTION;
```

DPR BACKGROUND FIXED DATA BUFFER (TURBO)

DPR Background Fixed Data Buffer Initialization and Handshaking (Turbo)

Startup/Shutdown and Handshaking Functions

```
BOOL PmacDPRBackgroundEx(DWORD dwDevice,int on,UINT period,UINT crd);
```

To enable this feature, call the PmacDPRBackgroundEx() routine with the **on** parameter set to a non-zero value, the **period** argument set anywhere from 1 to 255 (servo periods), and the **crd** parameter set from 1 – 8. To retrieve the information, use the provided DPR Background Fixed Data buffer query functions. Information for motors/coordinate systems 1- **crd** will be updated.
When enabled all the data query functions above may be used.
Typical usage of these routines is shown below, and also is available in the supplied example source code (PmacTest application).

DPR Background Fixed Data Buffer Query Routines (Turbo)

The “crd” / ”motor” parameter in the functions below is a 0 based index. For example, 0 is coordinate system 1, 1 is coordinate system 2 etc.

Data Query Functions

Motor Specific

```
double PmacDPRCommanded(DWORD dwDevice,long crd,char axchar);
```

This routine returns the commanded position for an axis (described by *axchar*) in coordinate system *crd* in the same units as used to define the axis.

```
double PmacDPRGetTargetPos(DWORD dwDevice,long motor,double posscale);
```

The target position of the specified *motor* is returned by this function. The units are in counts unless *posscale* is not unity.

Coordinate System Specific

```
long PmacDPRPe(DWORD dwDevice,long crd);
```

This function returns the program execution status for a given coordinate system, *crd*. The lower 24 bits are used. The low 24 bits are the same as the second word returned on a "??" command from PMAC.

```
long PmacDPRProgRemaining(DWORD dwDevice,long crd);
```

This function returns the number of program lines remaining for a specified coordinate system, *crd*. Same as the "PR" command for PMAC.

```
long PmacDPRTimeRemInMove(DWORD dwDevice,long crd);
```

This function returns the time remaining in the current move for a specified coordinate system, *crd*. The value this function returns is not very intuitive or useful for other than display purposes. The time for a single line in a program may be divided into three parts: acceleration, steady state, and deceleration times.

```
long PmacDPRTimeRemInTATS(DWORD dwDevice,long crd);
```

This function returns the time remaining in accel/decl when I13>0.

```
double PmacDPRGetFeedRateMode(DWORD dwDevice,int csn, BOOL *mode)
```

```
BOOL PmacDPRRotBufFull(DWORD dwDevice,long crd);
BOOL PmacDPRSysInposition(DWORD dwDevice,long crd);
BOOL PmacDPRSysWarnFError(DWORD dwDevice,long crd);
BOOL PmacDPRSysFatalFError(DWORD dwDevice,long crd);
BOOL PmacDPRSysRunTimeError(DWORD dwDevice,long crd);
BOOL PmacDPRSysCircleRadError(DWORD dwDevice,long crd);
BOOL PmacDPRSysAmpFaultError(DWORD dwDevice,long crd);
BOOL PmacDPRProgRunning(DWORD dwDevice,long crd);
BOOL PmacDPRProgStepping(DWORD dwDevice,long crd);
BOOL PmacDPRProgContMotion(DWORD dwDevice,long crd);
BOOL PmacDPRProgContRequest(DWORD dwDevice,long crd);

BOOL PmacDPRMotionBufOpen( DWORD dwDevice)
BOOL PmacDPRRotBufOpen( DWORD dwDevice )
```

The functions above return the boolean status of a variety of coordinate system flags.

Global

```
BOOL PmacDPRSysServoError( DWORD dwDevice )
BOOL PmacDPRSysReEntryError( DWORD dwDevice )
BOOL PmacDPRSysMemChecksumError( DWORD dwDevice )
BOOL PmacDPRSysPromChecksumError( DWORD dwDevice )
void PmacDPRGetGlobalStatus(DWORD dwDevice,VOID *gstatus)
```

Logical Query Functions

```
MOTIONMODE PmacDPRGetMotionMode(DWORD dwDevice,long cs);
```

The motion mode of a chosen coordinate system is returned by this function. The MOTIONMODE enumeration is shown below:

`typedef enum { linear,rapid,circw,circcw,spline,pvt } MOTIONMODE;`

```
PROGRAM PmacDPRGetProgramMode(DWORD dwDevice,long cs);
```

The program mode of a chosen coordinate system is returned by this function. The PROGRAMMODE enumeration is shown below:

```
typedef enum { stop,run,step,hold,joghold,jogstop } PROGRAM;
```

DPR VARIABLE BACKGROUND READ/WRITE DATA BUFFER

PmacDPRBackground()

```
BOOL PmacDPRBackground(DWORD dwDevice, long on);
```

Starts or stops the PMAC's automatic write into the DPR background fixed and variable buffer.

Arguments

dwDevice	Device number.
on	Turn on or off Boolean.

Returns

TRUE if success.

PmacDPRBufLast()

```
long PmacDPRBufLast(DWORD dwDevice);
```

This function determines which buffer is the last variable sized buffer in DPR.
Use this function to decide which buffer (either a binary rotary or the variable background data buffer) can be initialized or removed. The sequence for the binary rotary and variable background data buffers is like a last in first out stack. The order of initialization must be:

1. Binary rotary buffer 0
2. Binary rotary buffer 1
3. Variable background data buffer

Returns

<u>From 1.16B and beyond</u>

9 = Background Variable Buffer

8 = Binary Rotary Buffer #8

7 = Binary Rotary Buffer #7

6 = Binary Rotary Buffer #6

5 = Binary Rotary Buffer #5

4 = Binary Rotary Buffer #4

3 = Binary Rotary Buffer #3

2 = Binary Rotary Buffer #2

1 = Binary Rotary Buffer #1

0 = No Buffers in DPR

<u>Before 1.16B</u>

3 = Background Variable Buffer

2 = Binary Rotary Buffer #2

1 = Binary Rotary Buffer #1

0 = No Buffers in DPR

PmacDPRGetVBGAddress()

```
unsigned long PmacDPRGetVBGAddress(DWORD dwDevice, long handle, long entry_num);
```

This function returns an entry within the address table, initialized by PmacDPRVarBufInit(). For the first entry, set entry_num to 0; the second set entry_num to 1, etc. PMAC reads this address table to determine what to copy into DPR memory. Typically not used by programmers.

Arguments

dwDevice	Device number.
handle	Handle to the buffer.
entry_num	Data entry to be received from table, see for more detail

Returns

Returns the address table entry.

PmacDPRGetVBGNumEntries()

```
long  PmacDPRGetVBGNumEntries(DWORD dwDevice, long handle);
```

Returns the number of items being updated in the buffer by the calling application. This number is the same as that passed to the function PmacDPRVarBufInit().

Arguments

dwDevice	Device number.
handle	Handle to the buffer.

Returns

Returns the total number of entries.

PmacDPRGetVBGDataOffset()

```
UINT   PmacDPRGetVBGDataOffset(DWORD dwDevice, long handle);
```

Returns the offset, in bytes, from the beginning of DPR that the data for this applications VBGDB begins. Typically not used by programmers.

Arguments

dwDevice	Device number.

handle	Handle to the buffer.

Returns

Returns the offset in bytes.

PmacDPRGetVBGAddrOffset()

```
UINT   PmacDPRGetVBGAddrOffset(DWORD dwDevice, long handle);
```

Returns the offset, in bytes, from the beginning of DPR that the address array for this applications VBGDB begins. Divide by 4 to get the PMAC offset (from beginning of DPR 0xD000). Typically not used by programmers.

Arguments

dwDevice	Device number.
handle	Handle to the buffer.

Returns

Returns the offset in bytes.

PmacDPRGetVBGServoTimer()

```
UINT   PmacDPRGetVBGServoTimer(DWORD dwDevice);
```

Returns the value of the PMAC's servo timer

Arguments

dwDevice	Device number.

Returns

Returns the PMAC servo timer.

PmacDPRGetVBGStartAddr()

```
UINT PmacDPRGetVBGStartAddr(DWORD dwDevice);
```

Use this to attain the PMAC address of the start of the variable data buffer. This address also indicates the end of the free user memory.

Arguments

dwDevice	Device number.

Returns

The address of the start of the buffer.

PmacDPRGetVBGTotalEntries()

```
long   PmacDPRGetVBGTotalEntries(DWORD dwDevice);
```

Returns the number of items being updated in the buffer by all applications. Should never exceed 128.

Arguments

dwDevice	Device number.

Returns

Returns the total number of entries.

PmacDPRVarBufChange()

```
long PmacDPRVarBufChange(DWORD dwDevice, long handle, long num_entries, long *addrarray)
```

Changes a previously initialized multi-user background variable buffer (VBGDB).
User's VBGDB structure is allocated and updated with correct address offset and data offset. If other users are present, then their VBGDB structures will also be updated.
The buffer's start address and size will be initialized along with the address array in the dual ported RAM.

Assumptions

- That this buffer was the last one to be initialized in the DPR buffer area.
- That the DPR gather size is correct modulo of the number of variables and their size(32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48 bit PMAC variable. For example have 3 variables 2 x 24 and 1 x 48 which takes 4 x 32 bits. The size must be an exact modulo of 4, because the PMAC dual ported RAM gather does not check before storing that there is enough room remaining.
- That only one device structure is initialized per PMAC in the system

Arguments

dwDevice	Device number.
handle	Handle to previously initialized VBGDB.
num_entries	The size for address Buffer.
addrarray[]	Array of address and types.

Return Value

If it all works out a TRUE will be returned; otherwise FALSE.

PmacDPRVarBufInit()

```
long PmacDPRVarBufInit(DWORD dwDevice, long num_entries, long *addrarray);
```

Initializes a multi-user background variable buffer. (VBGDB)
User's VBGDB structure is allocated and updated with correct address offset and data offset. If other users are present, then their VBGDB structures will also be updated.
The buffer's start address and size will be initialized along with the address array in the dual ported RAM.

Assumptions

- That this buffer was the last one to be initialized in the DPR buffer area.
- That the DPR gather size is correct modulo of the number of variables and their size(32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48 bit PMAC variable. For example, have 3 variables 2 x 24 and 1 x 48, which takes 4 x 32 bits. The size must be an exact modulo of 4, because the PMAC dual ported RAM gather does not check before storing that there is enough room remaining.
- That only one device structure is initialized per PMAC in the system.

Arguments

dwDevice	Device number.
num_entries	The size for address Buffer.
addrarray[]	Array of address and types.

Return Value

A user handle if the Variable buffer is initialized OK; otherwise 0.

PmacDPRVarBufRead()

```
LONG PmacDPRVarBufRead(DWORD dwDevice, long h, long entry_num, long *long_2)
```

Reads an entry from the VBGD buffer and determines necessary offset to get the entry number desired.
Returns two longs by use of a pointer **long_2*.
Calling party should poll this function until TRUE is returned. Only then will the data be valid.

Arguments

dwDevice	Device number.
h	Handle to users VBGDB status structure.
entry_num	Entry number to be returned.
long_2	Pointers to returned data. Second element may or may not be valid. It's a function of data type for entry_num in question.

Return Value

Returns TRUE(1) on data ready else FALSE(0).

PmacDPRVarBufReadEx()

```
LONG PmacDPRVarBufReadEx(DWORD dwDevice, long h, void *long_x )
```

Reads an entry from the VBGD buffer and determines necessary offset to get the entry number desired. Copies all the data specified by the **addrarray** parameter of PmacDPRVarBufInit() routine to the long_x array. Therefore, the long_x array should be at least **num_entries** * 2 in size.
Calling party should poll this function until TRUE is returned. Only then will the data be valid. All necessary handshaking is handled within this routine.

Arguments

dwDevice	Device number.
h	Handle to users VBGDB status structure.
long_x	Pointers to returned data.. Data is packed tightly, last entry may not be valid.

Return Value

Returns TRUE(1) on data ready else FALSE(0).

PmacDPRVarBufRemove()

```
long PmacDPRVarBufRemove(DWORD dwDevice,long h);
```

Removes a user from the Multi-User Background Variable Buffer.

Arguments

dwDevice	Device number.
h	Handle to users VBGDB status structure.

Return Value

If successful, a TRUE(1) will be returned otherwise FALSE(0).

PmacDPRWriteBufferEx()

```
long PmacDPRWriteBufferEx(DWORD dwDevice, long num_entries, struct VBGWFormat *the_data);
```

This feature is available for PROM Version 1.15G and above.
The variable background write buffer can be used to have PMAC transfer up to 32 PMAC long or short words from DPR to its own internal memory without using the DPR ASCII feature. The data to be written into PMAC's memory is first placed in an array of VBGWFormat structures (definition shown below).

```
struct VBGWFormat{
        long type_addr;
        long data1;
        long data2;
};
```

The upper 16 bits of type_addr element specifies the type of data and the lower 16 bits specify the address to which data1 and data2 should be written to.
The types of data that may be specified are as follows.

Bits 0-2	0/1/2/4 for Y/L/X/SPECIAL memory respectively.
Bits 3-7	Width = 1,4,8,12,16,20 (a value of 0 is 24 bit variable).
Bits 8-12	Offset = 0..23.
Bits 13-15	Special type.

The way in which one uses data1, and data2 is not intuitive. For Y or X memory data1 element will be used for specifying the 24 bits of data. The most significant byte of data1 and all of data 2 are irrelevant in this case. For L data, PMAC 48 bit word, data1 should hold the first 32 bits and data2 should hold the most significant 16 bits leaving the most significant 16 bits of data2 irrelevant.
TWS is not yet implemented.
PMAC addresses are specified using an array of long integers. The most significant word of each long (upper 16 bits) specifies the word type. A value of 0, 1, 2 and 4 corresponds to Y, Long, X, and SPECIAL respectively. For Y, Long and X entries the least significant word specifies the actual PMAC address to be copied.

Arguments

num_entries	Number of elements in the the_data array to be used, max 32
the_data	A pointer to an array of VBGWFormat structures which specify what is copied from DPR into PMAC's internal memory.
entry_num	Data entry to be received from table, see above for more detail

DPR BINARY ROTARY BUFFER FUNCTIONS

NOTE: The *bufnum* parameter in the routines below is a 0 based index. Therefore, 0 would specify buffer/coordinate system 1, 1 would be buffer/coordinate system 2 etc.

PmacDPRAsciiFileToRot()

```
SHORT PmacDPRAsciiFileToRot(DWORD dwDevice, FILE *inpfile, USHORT bufnum);
```

Inputs ASCII strings from file "*inpfile*," converts all alpha to upper case, then converts the ASCII to PMAC commands and writes the data to the dual ported RAM binary rotary buffer when ready.

Arguments

dwDevice	Device number.
inpfile	NULL terminated path to file.
bufnum	Binary rotary buffer number.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons. When this occurs, the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRAsciiStrToRot()

```
SHORT PmacDPRAsciiStrToRotA(DWORD dwDevice, PCHAR inpstr, USHORT bufnum)
SHORT PmacDPRAsciiStrToRotW(DWORD dwDevice, PWCHAR inpstr, USHORT bufnum)
```

PmacDPRAsciiStrToRot() takes an ASCII Native PMAC text string, converts it to Native PMAC Binary, then places it into the DPR Binary Rotary Buffer if it has been set up and there is room.

Arguments

dwDevice	Device number.
inpstr	NULL terminated PMAC command string.
Bufnum	Binary rotary buffer number.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons. When this occurs, the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRAsciiStrToRotEx()

```
SHORT PmacDPRAsciiStrToRotEx(DWORD dwDevice, PCHAR inpstr, USHORT bufnum, BOOL
       bSendImmediately)
```

PmacDPRAsciiStrToRotEx() takes an ASCII Native PMAC text string, converts it to Native PMAC Binary, then places it into the DPR Binary Rotary Buffer if it has been set up and there is room.

Arguments

dwDevice	Device number.
inpstr	NULL terminated PMAC command string.
Bufnum	Binary rotary buffer number.
BSendImmediately	BOOL flag meant to send the data in one sweep. Use of this flag is only available for USB mode of communication for now and will be implemented in Ethernet mode of communication soon.

Return Value

Mnemonic	Returned Value	Explanation
IDS_ERR_059	-59	"RS274 to BIN DPROT Unable to allocate memory"
IDS_ERR_060	-60	"RS274 to BIN DPROT Unable to pack floating point number"
IDS_ERR_061	-61	"RS274 to BIN DPROT Unable to convert string to float number"
IDS_ERR_062	-62	"RS274 to BIN DPROT Illegal Command or Format in string"
IDS_ERR_063	-63	"RS274 to BIN DPROT Integer number out of range"
DprOk	0	The code was successfully sent to DPR
DprBufBsy	1	DPR Binary Rotary Buffer is Busy, please try again soon. Also, PMAC may stop running the program for a variety of reasons. When this occurs, the DPR Rotary Buffer will fill up and appear busy to the PC.
DprEOF	2	DPR Binary Rotary Buffer End of File detected

PmacDPRRotBufChange()

```
long PmacDPRRotBufChange(DWORD dwDevice, long bufnum,long new_size)
```

The PmacDPRRotBufChange() function is used to initialize the DPR Binary Rotary motion program buffers. Presently, there may be up to eight rotary buffers, buffer #0 through buffer #7 (All Turbo PMACs support 8; however, non-Turbo PMACs with firmware earlier than 1.16B will have only 2). Buffers #0, and #1 correspond to coordinate system 1 and 2, respectively. The binary rotary buffer you wish to initialize must be the last variable-sized buffer to be created. Therefore, if you wish to use 3 rotary buffers, begin with #0, then #1 and finally #2.
The size of a particular buffer is also specified with this function. Note that using PmacDPRRotBufChange() will not cause the change in size to be persistent when resetting the operating system (registry is not changed).

Note:

I57 should be set to 0 before changing the buffer size, and the buffer being changed must be the last variable-sized buffer initialized.

If *new_size* is set to zero, this function is equivalent to PmacDPRRotBufRemove().
Typically, the size of the rotary buffer in PMAC is half that allocated in the DP Ram. Two long words in the DP Ram (64 bits), are mapped into one long PMAC word (48 bits).
The value assigned to size, which is in units of 32 bits, specifies how many instructions will fit into the buffer. Each ASCII instruction is ultimately transformed into a 64-bit code (from which PMAC will transform into a 48-bit operation code). A single instruction consists of a program command and its argument. For example, "X100" is a single instruction and "X1000Y1000Z1000" is three instructions. A simple formula to determine the bufsize as a function of the maximum number of buffered program statements is:
buffer_size = (number_of_motors_in_cs) * (num_lines) * (2);

The Rotary buffer size must be at least 6 words long. If too large, it may run into the gather buffer.
Call PmacDPRRotBuf() to enable the Binary Rotary Buffer after ite has been initialized.

Arguments

dwDevice	Device number.
bufnum	Which of the two rotary buffers (i.e. 0 or 1) to reference.
new_size	Size of new buffer.

Non-Parameter Inputs:

Buffer Size

Registry value "DualPortRamRot(bufnum)" = the size for the desired buffer in (32 bit word units).
If the value of this key in the registry is 0, this effectively removes this rotary buffer. If greater than 0, this means you want to initialize #1, #2 etc. in DPR. The order must be rotary buffer #1 first then #2 etc. If the size has not changed, this will flush the buffer.
You can resize this buffer by calling "PmacDPRRotBufChange;" however, it must be the last buffer in DPR. To determine which was the last allocated buffer, use PmacDPRBufLast().

Notes

Variable sized buffers such as this one are added to the end of available buffer memory. This leaves room for the Gather buffer, which is fixed at the beginning of the DPR address space.

Assumptions

That the DPR gather size is a correct modulo of the number of variables and their size (32 bits for DPR for a 24-bit PMAC variable and 64 bits for a 48-bit variable).

Return Values

RET_ERROR (-1) Error (Check request, the buffer size was either too small or too large) RET_OK (0)

PmacDPRRotBufRemove()

```
long    PmacDPRRotBufRemove(DWORD dwDevice, long bufnum);
```

Use this function to remove any previously initialized rotary buffer.

Arguments

dwDevice	Device number.
Bufnum	Which of the two rotary buffers (i.e. 0 or 1) to reference.

Return Values

RET_ERROR (-1) (Check request, the buffer size was either too small or too large)

RET_OK (0)

PmacDPRRotBufClr()

```
void  PmacDPRRotBufClr(DWORD dwDevice, long bufnum);
```

This function will clear either Binary Rotary buffers 0 or 1 in DPR (i.e. remove all entries).

Arguments

dwDevice	Device number.
bufnum	Which of the two rotary buffers (i.e. 0 or 1) to reference.

Return Values

None

PmacDPRRotBuf()

```
long    PmacDPRRotBuf(DWORD dwDevice,BOOL onoff);
```

Once initialized with PmacDPRRotBufChange(), the PmacDPRotBuf() function can be used to enable or disable the rotary buffer (if **onoff** = 1 then enable if 0 then disable). The return value is a Boolean, TRUE for enabled, FALSE for disabled.
Internally, this routine sets I57 to the appropriate value, and also issues an “Open Rot” for non-Turbo PMACs or “Open Bin Rot” for Turbo PMACs.

Arguments

dwDevice	Device number.
onoff	Boolean value, Use 1 (ON) or 0 (OFF).

Return Values

1 = Active, 0 = Non-Active

DPR NUMERIC READ AND WRITE

General Information

The PmacDPRSet{DataType}() functions write numerical data to the specified **offset** while the PmacDPRGet{DataType}() reads at the specified **offset** and returns the data.
From within PMAC, data can be written to the DPR by use of PMAC M-variable assignments. Proper M-variable definitions for the corresponding data type are shown below:

Data Type	M variable Definition
16 bit integer	M{constant}>X/Y:{Address} (i.e. m100->X:$D200,0,16,s)
32 bit integer	M{constant}->DP:{Address} (i.e. m101->DP:$D201)
32 bit floating point	M{constant}->F:{Address} (i.e. m102->DP:$D202)

Standard Read/Write

```
WORD PmacDPRGetWord(DWORD dwDevice,UINT offset);
void    PmacDPRSetWord(DWORD dwDevice,UINT offset, WORD val);
DWORD PmacDPRGetDword(DWORD dwDevice,UINT offset);
void    PmacDPRSetDword(DWORD dwDevice,UINT offset, DWORD val);
float PmacDPRGetFloat(DWORD dwDevice,UINT offset);
void    PmacDPRSetFloat(DWORD dwDevice,UINT offset,float val);
```

Masking

```
BOOL PmacDPRDWordBitSet(DWORD dwDevice,UINT offset,UINT bit);
void PmacDPRSetDwordBit(DWORD dwDevice,UINT offset,UINT bit);
void PmacDPRResetDwordBit(DWORD dwDevice,UINT offset,UINT bit);
void PmacDPRSetDwordMask(DWORD dwDevice,UINT offset,DWORD val,BOOL onoff);
DWORD PmacDPRGetDwordMask(DWORD dwDevice,UINT offset,DWORD val);
```

Dual Word Conversion

```
double PmacDPRFloat(DWORD dwDevice,long d[],double scale);
double PmacDPRLFixed(DWORD dwDevice,long d[],double scale);
```

The "Dual Word" Conversion functions convert data that is placed in DPR by one of it's automatic features. Whenever a long word in PMAC (48 bit) is placed in DPR (Motor 1 actual position register for

example) each 24-bit short word (X and Y) is sign extended and placed in a 32-bit word, making it 64 bits of data that need to be converted.
A typical sequence of function calls for retrieving and converting motor 1 actual position, for example, would look like:

```
/* Get the dual word data */
long data[2];

While(1){
   /* Get second entry*/
   if(PmacDPRVarBufRead(dwDevice,myVBGBHandle,1,data)
)
   {
      /* Convert to encoder counts (Assuming I108 =
96)*/
      MotorPosition =

PmacDPRLFixed(dwDevice,data,1.0/(96.0*32.0));
      cprintf("\n\n Actual motor position after
conversion           =%lf\r\n",MotorPosition);
   }
}
```

The second parameter passed to PmacDPRLFixed() is the scale factor. Since this position register in units of (32*I108) encoder counts, the inverse of this is used to scale the return value.

PmacDPRDWordBitSet()

```
BOOL PmacDPRDWordBitSet(DWORD dwDevice, UINT offset, UINT bit);
```

Determines if a bit is set within a 32-bit word (DWORD) in dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
bit	Bit number to set. 0..31.

Return Values

TRUE if bit = 1.

PmacDPRFloat()

```
double PmacDPRFloat(long d[],double scale);
```

Converts a 48-bit floating point word: 36-bit mantissa / 12 exponent packed into two 32-bit words holding 24 bits each of the 48-bit words. Multiplies the result by “*scale*.”

Arguments

d[]	Two 32 bit long values to converts.
scale	Scale multiplier.

Return Values

Double floating point representation of the 48-bt number.

PmacDPRGetDWord()

```
DWORD PmacDPRGetDWord(DWORD dwDevice,UINT offset);
```

Retrieves a 32-bit DWORD from dual ported RAM.

Arguments

dwDevice	Device number.
Offset	Offset from the start of dual ported RAM.

Return Values

DWORD at offset location.

PmacDPRGetDWordMask()

```
DWORD   PmacDPRGetDWordMask(DWORD dwDevice, UINT offset, DWORD mask);
```

Retrieves a double word from dual ported RAM memory at the **offset**, masks (bitwise AND) it with **val** and then returns the resulting value.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
mask	Bitwise and mask used with value read from dual ported RAM.

Return Values

Result of DWORD and mask at offset location.

PmacDPRGetFloat()

```
float   PmacDPRGetFloat(DWORD dwDevice, UINT offset);
Reads an IEEE 32-bit floating point value from dual ported RAM.
```

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.

Return Values

Float at offset location.

PmacDPRGetMem()

```
PVOID PmacDPRGetMem(DWORD dwDevice, DWORD offset, size_t count, PVOID val);
```

Copies a block of dual ported RAM memory.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
count	Size of memory block to copy.
val	Pointer to destination.

Return Values

Pointer to destination.

PmacDPRGetWord()

```
WORD PmacDPRGetWord(DWORD dwDevice, UINT offset);
```

Retrieves a 16-bit word from dual ported ram.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.

Return Values

Pointer to destination.

PmacDPRResetDWordBit()

```
void PmacDPRResetDWordBit(DWORD dwDevice, UINT offset, UINT bit);
```

Exclusive XORS a bit within a 32-bit word (DWORD) in dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
bit	Bit number to set. 0..31.

Return Values

None.

PmacDPRSetDWord()

```
void PmacDPRSetDWord(DWORD dwDevice, UINT offset, DWORD val);
```

Sets the value of a 32-bit word in dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
val	Value to write to DPRAM.

Return Values

None.

PmacDPRSetDWordBit()

```
void PmacDPRSetDWordBit(DWORD dwDevice, UINT offset, UINT bit);
```

Turns on a bit (sets to one(1)) within a 32-bit word (DWORD) in dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
bit	Bit number to set. 0..31.

Return Values

None.

PmacDPRSetDWordMask()

```
void PmacDPRSetDWordMask(DWORD dwDevice, UINT offset, DWORD val, long onoff);
```

This function is used to manipulate DPR memory a DWORD (32 bits) at time. A bit wise logical OR of the register specified by **offset** is done with **val** and then stored into the same specified DPR location. If **onoff** is TRUE the a bit wise exclusive or XOR is evaluate using **val** and the specified memory location. The result of this operation is then used to perform a bit wise AND with the value located in DPR. The result is stored in the specified DPR location.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
val	Value to store.
onoff	Boolean indicating exclusive XOR or AND of *val* with DPRAM memory.

Return Values

None.

PmacDPRSetFloat()

```
void PmacDPRSetFloat(DWORD dwDevice, UINT offset, double val);
```

Writes a floating point value into dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
val	Value to store.

Return Values

None.

PmacDPRSetMem()

```
PVOID PmacDPRSetMem(DWORD dwDevice, DWORD offset, size_t count, PVOID val);
```

Copies a block of memory into dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
count	Size of data to transfer.
val	Pointer to memory to transfer.

Return Values

Pointer to transferred data.

PmacDPRSetWord()

```
void PmacDPRSetWord(DWORD dwDevice, UINT offset, WORD val);
```

Writes a 16-bit word value into dual ported RAM.

Arguments

dwDevice	Device number.
offset	Offset from the start of dual ported RAM.
val	Value to store.

Return Values

None.

DATA GATHERING FUNCTIONS

PmacSetGather

```
BOOL PmacSetGather( DWORD dwDevice,UINT num,LPSTR str,BOOL ena );
```

Sets the gather address and number of samples to gather.

Arguments

dwDevice	Device number.
num	Number of samples to gather.
str	Base address of each sample to gather.
ena	Enable or disable the gather function.

Return Values

TRUE = Ok, FALSE = Error.

PmacSetQuickGatherEx

```
BOOL PmacSetQuickGatherEx( DWORD dwDevice,PWTG_EX mask,BOOL ena );
```

This function takes a mask as described in Gather.h and configures the gather to collect from a mask of these standards. This is for convienence and ease of use.

Arguments

dwDevice	Device number.
mask	mask . See gather.h for details.

```
typedef struct _WTG_EX
{
  UINT COM_TO_G;
  UINT ENC_TO_G;
  UINT DAC_TO_G;
  UINT CUR_TO_G;
 } WTG_EX, *PWTG_EX;
```

ena	TRUE or FALSE Enable or Disable QuickGather.

Return Values

TRUE = Ok, FALSE = Error.

PmacGetGather

```
BOOL PmacGetGather( DWORD dwDevice,UINT num,LPSTR str,UINT maxchar );
```

Arguments

dwDevice	Device number.
`num`	Number of samples to gather.
`str`	.Base address of each sample to gather.
`maxchar`	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

PmacStartGather

```
int PmacStartGather( DWORD dwDevice )
```

Arguments

dwDevice	number.

Return Values

TRUE =1.

PmacStopGather

```
int PmacStopGather( DWORD dwDevice )
```

Arguments

dwDevice	number.

Return Values

TRUE = 1

PmacSetQuickGatherWithDirectCurrentEx

```
BOOL PmacSetQuickGatherWithDirectCurrentEx( DWORD dwDevice, PWTG_EX mask, BOOL ena );
```

Arguments

dwDevice	Device number.
mask	mask . See gather.h for details.

```
typedef struct _WTG_EX
{
  UINT COM_TO_G;
  UINT ENC_TO_G;
  UINT DAC_TO_G;
  UINT CUR_TO_G;
 } WTG_EX, *PWTG_EX;
```

ena	TRUE or FALSE Enable or Disable.

Return Values

TRUE = Ok, FALSE = Error.

INTERRUPT FUNCTIONS

General Information Interrupt Functions

Interrupts are provided for both Windows 95/98 and Windows NT operating systems. There are four methods of interrupt notification for your program:

1. Send a message to a window.
2. Call a function.
3. Start a thread function.
4. Set an event.

Each method has a separate initialization function. Common to all initialization functions is the parameter *ulMask*. This parameter determines the interrupt service vector(s) to be used for the interrupt initiated by the function.

The least significant byte of *ulMask* controls which conditions will generate an interrupt. A bit value of 0 enables, 1 disables.

Bit	PMAC Signal
0	In Position of Coordinate System
1	Buffer Request (PMAC's request for more moves)
2	Error, A motor(s) in the coordinate system has had a fatal following error
3	Warning, A motor(s) in the coordinate system has had a warning following error
4	Host Request, PMAC has an ASCII response for the host
5-7	User programmable, see PMAC User's Guide, Writing a Host Communications Program

PmacINTRFireEventInit()

```
BOOL  PmacINTRFireEventInit(DWORD dwDevice,HANDLE hEvent,ULONG ulMask);
```

Once initialized with PmacINTRFireEventInit(), an interrupt will cause the driver to set the event handle provided to the signaled state. The programmer is responsible to reset the event to the non-signaled state.

Arguments

dwDevice	Device number.
hEvent	Handle to the event to be set.
ulMask	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

PmacINTRFuncCallInit()

```
BOOL  PmacINTRFuncCallInit(DWORD dwDevice,PMACINTRPROC pFunc,DWORD msg,ULONG ulMask);
```

Once initialized with PmacINTRFuncCallInit(), an interrupt will cause the driver to call a function provided of PMACINTRPROC type.

Arguments

dwDevice	Device number.
pFunc	Pointer to a PMACINTRPROC function.
msg	Message to be passed back to the called function as a parameter.
ulMask	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

PmacINTRRunThreadInit()

```
BOOL  PmacINTRRunThreadInit(DWORD dwDevice,LPTHREAD_START_ROUTINE pFunc,UINT msg,ULONG
       ulMask);
```

Once initialized with PmacINTRRunThreadInit(), an interrupt will cause the driver to initialize a thread function provided of LPTHREAD_START_ ROUTINE type. The programmer is responsible for synchronization and termination of the thread.

Arguments

dwDevice	Device number.
pFunc	Pointer to a LPTHREAD_START_ROUTINE function.
msg	Message to be passed back to the called function as part of a parameter structure.
ulMask	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

PmacINTRTerminate()

```
BOOL  PmacINTRTerminate(DWORD dwDevice);
```

Shuts down the interrupt service.

Arguments

dwDevice	number.

Return Values

TRUE = Ok, FALSE = Error.

PmacINTRWndMsgInit()

```
BOOL  PmacINTRWndMsgInit(DWORD dwDevice,HWND hWnd,UINT msg,ULONG ulMask);
```

Once initialized with PmacINTRWndMsgInit(), an interrupt will cause the driver to send provided message to the provided windows handle.

Arguments

dwDevice	Device number.
hWnd	Window handle.
msg	Message to be sent to the window.
ulMask	32-bit interrupt mask. Zero(0) will cause all interrupt service levels to interrupt.

Return Values

TRUE = Ok, FALSE = Error.

SERIAL CHECKSUM COMMUNICATION FUNCTIONS

```
long PmacSERDoChecksums(DWORD dwDevice, UINT do_checksums);
long PmacSERCheckResponseA(DWORD dwDevice, char * response, UINT maxchar,char * outchar);
BOOL PmacSERCheckSendLineA(DWORD dwDevice,char * outchar, char * command_csum);
long PmacSERCheckGetLineA(DWORD dwDevice,PCHAR response,UINT maxchar, PUINT num_char);
```

Note:

See Terminal.C example program for a terminal example of using checksums.

PmacSERDoChecksums()

To enable or disable serial checksummed communications call PmacSERDoChecksums(). A non-zero return value indicates a successful mode change.
Once enabled, the last three routines above may be called. By far the easiest and most useful is the PmacSERCheckResponseA() routine.

PmacSERCheckResponseA()

PmacSERCheckResponseA() is used to provide checksum verification of commands sent to PMAC. It is capable of retrieving both multiple and single line responses. This function returns the total number of characters received from PMAC, including checksum characters. Since the checksum characters are removed from the response, the final number of characters in the response, and the value return by this function will differ. In any event, if the value returned by this function becomes equal to the "maxchar" parameter PMAC's total response was not received. If this occurs, use SERflush to clear out PMAC's buffer, increase the size of the character array (and also maxchar) and try again. If a an error occurs, this function will return FALSE.
This function uses the "control-n" feature implemented in PROM versions 1.14 and beyond. This feature enables the host to assure that PMAC has received the command correctly before sending the carriage return which tells PMAC to act on the just sent command.

Note:

If PMAC's response is larger than "maxchar," the remainder of the response will be left in the rotary buffer. This can cause garbled communications; therefore, always oversize the "response" buffer and the "maxchar" specification.

The following error codes may be set after calling this routine:

IDS_CHECKSUMDATABAD	Checksum communications error Bad Data.
IDS_CHECKSUMACKBAD	Checksum communications error Bad Acknowledge

PmacSERCheckSendLineA()

Function: This function sends a line to PMAC but first checks to see if it got down correctly. It returns TRUE on successful transmission of the command. The command sum for the command is also passed back by value. See below for an example of it's usage within the PmacSERCheckResponseA() routine.

PmacSERCheckGetLineA()

This function is used to retrieve and checksum verify a response from PMAC. For multiline responses this routine pulls out one line at a time. A line is delimited by a CR. It checks each lines Data Checksum. When the ACK comes at the end, the command checksum is placed in *response[0]* and COMM_EOT is returned.
If either a checksum, or a timeout occurs, COMM_FAIL is returned and the appropriate error code set (call GetError(), GetErrorStr() to retrieve the error). See below for an example of it's usage within the PmacSERCheckResponseA() routine.

Source/Example PmacSERCheckResponse()

```
long PmacSERCheckResponse(DWORD dwDevice,char *
response, UINT maxchar,char * outchar)
{
      char *cp, csum, command_sum_from_PMAC;
      DWORD nc =0;
      int      ic,attempt,ret;
      DWORD tc;
  response[0]=0;
  si.SERCommError = ERR_OK;
  attempt = 0;
  if(si.Initialized == FALSE) return 0;
  if(strlen(outchar) == 0)
    return 0;
  while(attempt < NUM_CHECKSUM_RETRIES){
    if(outchar[0] <= 26)// It's a control char
    {
      csum = outchar[0];
      PmacSERSendCharA(dwDevice,outchar[0]);
    }
    else
if(!PmacSERCheckSendLine(dwDevice,outchar,&csum))
      goto error;
    cp = response;
        tc = maxchar;
        while(tc > 0 && (ret =
PmacSERCheckGetLine(dwDevice,cp,tc,&ic)) >= 0)
    {
            tc -= (ic +1);
              cp += ic;
    }
```

```
        if(IS_COMM_FAIL(ret))
        {
        //            si.SERCommError =
          IDS_CHECKSUMDATABAD;
        // compare data checksum
        //    si.SERCommError = COMM_TIMEOUT;
          goto error;
        }
        if(IS_COMM_EOT(ret))
        {
          command_sum_from_PMAC = *cp;
          *cp = 0;// Remove the checksum from the
    response.
          if(command_sum_from_PMAC == !csum)
          { // IS IT AN ACKNOWLEDGE
            si.SERCommError = IDS_CHECKSUMACKBAD;
                    goto error;
          }
        }
        if(IS_COMM_ERROR(ret))
                      goto error;
        if(nc == maxchar)
        {
          si.SERCommError = IDS_INCOMP_RECEIVE;
          // Unable to get complete response
          goto error;
        }
          // Remove checksums from response
        nc = (maxchar - tc);
        return nc;
        error:
        attempt++;
        SERFlush(dwDevice);
    }
    return 0;
}
```

VARIABLE FUNCTIONS

PmacGetVariable()

```
short int PmacGetVariable(DWORD dwDevice,char ch,UINT num,short long def);
long PmacGetVariableLong(DWORD dwDevice,char ch,UINT num,long def);
double PmacGetVariableDouble(DWORD dwDevice,char ch,UINT num,double def);
```

Returns a PMAC variable (I,M,P or Q) value as a 16-bit integer, long 32-bit integer or 80-bit floating point double. If there is a communication error, then the default value in parameter 'def' will be returned.

Arguments

dwDevice	Device number.
Ch	Character representing the variable type. 'M', 'P', 'I', 'Q'.
num	Variable number to get.
def	Default value.

Return Value

'I' variable value or default.

PmacSetVariable()

```
void PmacSetVariable(DWORD dwDevice, char ch, UINT num, short int val);
void PmacSetVariableLong(DWORD dwDevice, char ch, UINT num, long val);
void PmacSetVariableDouble(DWORD dwDevice, char ch, UINT num, double val);
```

Sets a PMAC variable value as a 16bit integer, long 32bit integer or 80bit floating point double.

Arguments

dwDevice	Device number.
Ch	Character representing the variable type. 'M', 'P', 'I', 'Q'.
num	Variable number.
val	Value to be set.

Return Value

None.

UTILITY FUNCTIONS

PmacGetRomDateA()

```
PCHAR PmacGetRomDateA(DWORD dwDevice,PCHAR s,long maxchar);
```

Fills a string buffer with the firmware date of the PMAC device.

Arguments

dwDevice	Device number.
s	Pointer to string buffer.
Maxchar	Maximum characters to copy.

Return Value

Pointer to string buffer.

PmacGetRomVersionA()

```
PCHAR PmacGetRomVersionA(DWORD dwDevice,PCHAR s,long maxchar);
```

Fills a string buffer with the firmware version of the PMAC device.

Arguments

dwDevice	Device number.
s	Pointer to string buffer.
Maxchar	Maximum characters to copy.

Return Value

Pointer to string buffer.

PmacGetUserHandle()

```
PUSER_HANDLE PmacGetUserHandle(DWORD dwDevice);
```

This function is used to retrieve a handle to a communication information structure used internally by the driver. It is very similar to the PCOMM Library tagPmac structure. You will probably never need to use it.

Arguments

dwDevice	Device number.

Return Value

NULL if the function fails. Otherwise a pointer to USER_HANDLE.

PmacGetPmacType()

```
long PmacGetPmacType(DWORD dwDevice);
```

Returns an integer indicating the PMAC model you are communicating with.

Arguments

dwDevice	Device number.

Return Value

1 = PMAC1.

2 = PMAC2.

3 = PMAC Ultralite

5 = Turbo PMAC1

6 = Turbo PMAC2

7 = Turbo PMAC Ultralite

Inp()

```
BYTE Inp(DWORD dwDevice, BYTE offset);
```

Allows the user to get at PMAC's I/O registers, (BUSBASE + x). Useful for checking PMAC state without having to use ASCII communications. Other utility functions are provided that do similar things, but use ASCII communications.

Arguments

dwDevice	Device number
offset	Offset from PMAC's bus base address

Return Value

PMAC's Interrupt Status Register State if successful
0xFF if a failure occurred, Either communication not established or BUS or DPR communication is not being used

Assumptions

- Either the bus or DPR is being used. No good for serial port.
- If checking InPosition or other similar data bits from the status register (base + 2), the appropriate Motor or Coordinate system has been addressed.

Outp()

```
BOOL Outp(DWORD dwDevice, BYTE offset, BYTE outch);
```

Allows the user to set PMAC's I/O registers, (BUSBASE + x). Useful for getting PMAC out of a reset state for example. Either the Bus or DPR must be in use for this routine to succeed

Arguments

dwDevice	Device Number (PMAC 1,2,3 etc.)
offset	Offset from PMAC's bus base address
outch	value to be written to port

Return Value

TRUE if successful, else FALSE

Assumptions

Either the bus or DPR is being used. No good for serial port.

getBitValue()

```
long getBitValue(char *hexadecimal_number, long bit_requested);
```

Gets the value of a bit from a hexadecimal string.

Arguments

hexadecimal_number	Zero terminated hex string.
bit_requested	Bit number to examine.

Return Value

One(1) or zero(0).

hex_long2()

```
long hex_long2(char *in_str, long str_ln);
```

Converts an hexadecimal string to a long integer.

Arguments

in_str	Pointer to the beginning of hex string.
str_ln	How many characters to examine.

Return Value

Long integer value representation of hex string.

DATA TYPES, STRUCTURES, CALLBACKS, AND CONSTANTS

DOWNLOADGETPROC

```
typedef long  (FAR WINAPI *DOWNLOADGETPROC)(long nIndex,LPTSTR lpszBuffer,long nMaxLength);
```

This function type is used by some program downloading procedures to offer the option of extracting text lines for the download from another source other than a disk file.

Arguments

nIndex	Line number asked for.
lpszBuffer	Buffer pointer to copy text line into.
nMaxLength	Maximum length of buffer.

Return Value

The number of characters copied into the buffer.

DOWNLOADPROGRESS

```
typedef void (FAR WINAPI *DOWNLOADPROGRESS)(long nPercent);
```

This function type is used by some program downloading procedures to offer the option of setting the current percent of progress during the procedure.

Arguments

nPercent	Current percent of progress.

Return Value

None.

DOWNLOADMSGPROC

```
typedef void (FAR WINAPI *DOWNLOADMSGPROC)(LPTSTR str,BOOL newline);
```

This function type is used by some program downloading procedures to offer the option of reporting a status message.

Arguments

str	Message string.
newline	Indicates if a new line should be added by the called procedure.

Return Value

None.

DPRTESTMSGPROC

```
typedef void (FAR WINAPI *DPRTESTMSGPROC)(LONG NumErrors,LPTSTR Action,LONG CurrentOffset);
```

This function type is used by some program testing DPR via the PmacDPRTest() routine to offer the option of reporting the number of errors that have occurred, a status message, and the current offset being tested. The *action* message indicates to the user what action is actively being done in the testing of the DPR. The *action* message will also indicate to the user what errors occurred during the testing.

Action
"Setting up for Dual Ported Ram Test"
"Cannot set communications mode to BUS."
"Writing to DPR address (via PMAC)"
"Reading/Verifying DPR (via PC)"
"Writing to DPR Address (via PC)"
"Reading/Verifying DPR (via PMAC)"
"Clearing all Dual Ported Ram"

Arguments

NumErrors	Number of accumulated errors that have occurred
Action	Message indicating what part of test is being done
CurrentOffset	Offset from host base address being tested (in bytes)

Return Value

None.

DPRTESTPROGRESS

```
typedef void (FAR WINAPI *DPRTESTPROGRESS)(LONG Percent);
```

This function type is used by some program testing DPR via the PmacDPRTest() routine to offer the option of reporting a status message. The message indicates to the user what percent the test procedure is done with the current action being done. Actions include the following:

Action
"Setting up for Dual Ported Ram Test"
"Cannot set communications mode to BUS."
"Writing to DPR address (via PMAC)"
"Reading/Verifying DPR (via PC)"
"Writing to DPR Address (via PC)"
"Reading/Verifying DPR (via PMAC)"
"Clearing all Dual Ported Ram"

Arguments

Percent	Percent done with current action being done in test

Return Value

None.

Dpr_Bin_Rot_Errors

Used by binary rotary buffer functions.

```
enum Dpr_Bin_Rot_Errors
{
  DprOk         =   0,
  DprBufBsy     =   1,      // DPR Rotary Buffer is Busy
  DprEOF        =   2,      // DPR Rotary Buffer End of File
                            detected
  DprFltErr     =  -1,      // Unable to pack; floating point
                            number too large
  DprStrToDErr  =  -2,      // Unable to convert string to
                            double float number
  DprMtrErr     =  -3,      // Unable to convert Motor numbers
                            specified
  DprPlcErr     =  -4,      // Unable to convert PLC numbers
                            specified
  DprAxisErr    =  -5,      // Unable to convert axis in ABS,
                            INC or FRAX
  DprCmdErr     =  -6,      // Illegal Command in string
  DprCmdMaxErr  =  -7,      // Exceeded the Max number of PMAC
                            cmds in a line  ( See DPRCMDMAX )
  DprIntErr     =  -8,      // Integer number out of range (
                            See  Integer Max values CIRMAX,
                            etc. )
  DprBufErr     =  -9,      // DPR Rotary or Internal Rotary
                            Buffer Size is zero
  DprOutFileErr = -10,      // DPR Output File Error
  DprInpFileErr = -11       // DPR Input File Error
};
```

GLOBALSTATUS

Used in DPR Real Time Buffer query routines

```
typedef struct gs { // Global Status
     // DWord 1 ( ??? 1st 24/32 bit word )
     USHORT rffu2                  : 8; // 0-7
     USHORT internal1      : 3; // 8-10
     USHORT buffer_full       : 1;
     USHORT internal2      : 3; // 12-14
     USHORT dpram_response : 1;
     USHORT plc_command       : 1;
     USHORT plc_buf_open   : 1;
     USHORT rot_buf_open   : 1; // 18
     USHORT prog_buf_open  : 1; // 19
     USHORT internal3      : 2;
     USHORT host_comm_mode : 1;
     USHORT internal4      : 1;
     USHORT pad2                   : 8;
     // DWord 2 ( ??? 2nd 24/32 bit word )
     USHORT rffu1                  : 7;
     USHORT end_gather          : 1;
     USHORT rapid_m_flag   : 1;
     USHORT rti_warning       : 1;
     USHORT earom_error       : 1;
     USHORT dpram_error       : 1;
     USHORT prom_checksum  : 1;
     USHORT mem_checksum   : 1;
     USHORT comp_on              : 1;
     USHORT stimulate_on   : 1;
     USHORT stimulus_ent   : 1;
     USHORT prep_trig_gat  : 1;
     USHORT prep_next_serv : 1;
     USHORT data_gat_on       : 1;
     USHORT servo_err      : 1;
     USHORT servo_active   : 1;
     USHORT intr_reentry   : 1;
     USHORT intr_active       : 1;
     USHORT pad1                   : 8;
} GLOBALSTATUS;
```

MOTION

typedef enum { inpos,jog,running,homing,handle,openloop,disabled } MOTION;

MOTIONMODE

typedef enum { linear,rapid,circw,circcw,spline,pvt } MOTIONMODE;

PROGRAM

typedef enum { stop,run,step,hold,joghold,jogstop } PROGRAM;

SERVOSTATUS

Used in DPR Real Time Buffer query routines

```
typedef struct ss { // Motor Servo Status ( ?  1st 24 bit word  )
          USHORT internal1              : 8;
          USHORT internal2      : 2;
          USHORT home_search    : 1;
          USHORT block_request          : 1;
          USHORT rffu1          : 1;
          USHORT desired_vel_0          : 1;
          USHORT data_block_err         : 1;
          USHORT dwelling       : 1;
          USHORT integration    : 1;
          USHORT run_program    : 1;
          USHORT open_loop              : 1;
          USHORT phased_motor           : 1;
          USHORT handwheel_ena          : 1;
          USHORT pos_limit              : 1;
          USHORT neg_limit              : 1;
          USHORT activated              : 1;
          USHORT pad            : 8;
} SERVOSTATUS;
```

EXTENDED FUNCTIONS

The following functions are extended versions of the existing functions (without the extension “**Ex**”). The “**Ex**” signifies the enhancement to the legacy code by their ability to get the actual status bit from the PMAC in addition to the number of characters. Furthermore, these functions have eliminated the existing bugs associated with the legacy code.
The purpose of using these new functions is to get the actual status code instead of the number of characters as was the case of legacy code (without Ex). The main difference from the old functions to the new ones is the return value. In the existing code the number of characters is returned. The new functions are written to get the error status in addition if any and are explained in detail in the following section.
It is strongly recommended that users use these new functions instead of the old functions.

PmacDPRRealTimeEx

```
BOOL PmacDPRRealTimeEx( DWORD dwDevice, long mask, UINT period, long on )
```

Used to enable the DPR RealTime data buffer for PMAC Types only

```
{
  char str[20 ];
  if (vh[dwDevice].DPRAM == NULL)
    return FALSE;
  if (gh[dwDevice].bIsTurbo)
  {
    // Turbo
    if (on == TRUE)
    {
      wsprintfA( str, "i47=%u", period );
      PmacSendCommandA( dwDevice, str );
      PmacSendCommandA( dwDevice, "i48=1" );
      vh[dwDevice].DPRRTBufferTurbo->hostbusy = 0;
      // Reset host busy
      vh[dwDevice].DPRRTBufferTurbo->dataready = 0;
      // Reset Data Ready bit
      gh[dwDevice].DPRRealtActive = TRUE;
    }
    else
    {
      PmacSendCommandA( dwDevice, "i47=0" );
      PmacSendCommandA( dwDevice, "i48=0" );
      gh[dwDevice].DPRRealtActive = FALSE;
```

```
          vh[dwDevice].DPRRTBufferTurbo->dataready = 0;
          // Reset Data Ready bit
        }
        vh[dwDevice].DPRRTBufferTurbo->motor_mask = mask;
      }
      else
      {
        if (on == TRUE)
        {
          wsprintfA( str, "i19=%u", period );
          PmacSendCommandA( dwDevice, str );
          PmacSendCommandA( dwDevice, "i48=1" );
          PmacSendCommandA( dwDevice, "gat" );
          // turn on real time buffer gather
          vh[dwDevice].DPRRTBuffer->hostbusy = 0;
          // Reset host busy
          gh[dwDevice].DPRRealtActive = TRUE;
        }
        else
        {
          PmacSendCommandA( dwDevice, "endgat" );
          // turn off     real time buffer gather
          PmacSendCommandA( dwDevice, "i48=0" );
          gh[dwDevice].DPRRealtActive = FALSE;
        }
      }
      return gh[dwDevice].DPRRealtActive;
    }
```

PmacDPRBackgroundEx()

`BOOL PmacDPRBackgroundEx(DWORD dwDevice, long on);`

Starts or stops the PMAC's automatic write into the DPR background fixed and variable buffer.

Arguments

dwDevice	Device number.
On	Turn on or off Boolean.

Returns

TRUE if successful.

PmacDPRVarBufInitEx()

`long PmacDPRVarBufInitEx(DWORD dwDevice, long num_entries, long *addrarray, PUINT addrtype);`
Initializes a multi-user background variable buffer. (VBGDB)
User's VBGDB structure is allocated and updated with correct address offset and data offset. If other users are present, then their VBGDB structures will also be updated.
The buffer's start address and size will be initialized along with the address array in the dual ported RAM.

Assumptions

- That this buffer was the last one to be initialized in the DPR buffer area.
- That the DPR gather size is correct modulo of the number of variables and their size(32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48 bit PMAC variable. For example have 3 variables 2 x 24 and 1 x 48 which takes 4 x 32 bits. So the size must be an exact modulo of 4 because the PMAC dual ported RAM gather does not check before storing that there is enough room remaining.
- That only one device structure is initialized per PMAC in the system.

Arguments

dwDevice	Device number.
num_entries	The size for address Buffer.
addrarray[]	Array of address and types.
addrtype:	pointer to type array that is used with addrarray to construct an address format that PMAC can handle.

Return Value

A user handle if the Variable buffer is initialized OK; otherwise 0.

PmacDPRVarBufChangeEx()

`long PmacDPRVarBufChangeEx(DWORD dwDevice, long handle, long num_entries, long *addrarray)`

Changes a previously initialized multi-user background variable buffer (VBGDB).
User's VBGDB structure is allocated and updated with correct address offset and data offset. If other users are present, then their VBGDB structures will also be updated.
The buffer's start address and size will be initialized along with the address array in the dual ported RAM.

Assumptions

- That this buffer was the last one to be initialized in the DPR buffer area.
- That the DPR gather size is correct modulo of the number of variables and their size (32 bits for DPR for a 24 bit PMAC variable and 64 bits for a 48 bit PMAC variable. For example have 3 variables 2 x 24 and 1 x 48 which takes 4 x 32 bits. So the size must be an exact modulo of 4 because the PMAC dual ported RAM gather does not check before storing that there is enough room remaining.
- That only one device structure is initialized per PMAC in the system.

Arguments

dwDevice	Device number.
handle	Handle to previously initialized VBGDB.

num_entries	The size for address Buffer.
addrarray[]	Array of address and types.

Return Value

If successful, a TRUE will be returned; otherwise FALSE.

PmacDPRWriteBufferEx()

```
long PmacDPRWriteBufferEx(DWORD dwDevice, long num_entries, struct VBGWFormat *the_data);
```

This feature is available for PROM Version 1.15G and above.
The variable background write buffer can be used to have PMAC transfer up to 32 PMAC long or short words from DPR to its own internal memory without using the DPR ASCII feature. The data to be written into PMAC's memory is first placed in an array of VBGWFormat structures (definition shown below).

```
struct VBGWFormat{
      long type_addr;
      long data1;
      long data2;
};
```

The upper 16 bits of type_addr element specify the type of data and the lower 16 bits specify the address to which data1 and data2 should be written to.
The types of data that may be specified are as follows.

Bits 0-2	0/1/2/4 for Y/L/X/SPECIAL memory respectively.
Bits 3-7	Width = 1,4,8,12,16,20 (a value of 0 is 24 bit variable).
Bits 8-12	Offset = 0..23.
Bits 13-15	Special type.

The way in which one uses data1 and data2 is not intuitive. For Y or X memory, data1 element will be used for specifying the 24 bits of data. The most significant byte of data1 and all of data 2 are irrelevant in this case. For L data, PMAC 48 bit word, data1 should hold the first 32 bits and data2 should hold the most significant 16 bits, leaving the most significant 16 bits of data2 irrelevant.
TWS is not yet implemented.
PMAC addresses are specified using an array of long integers. The most significant word of each long (upper 16 bits) specifies the word type. A value of 0, 1, 2 and 4 corresponds to Y, Long, X, and SPECIAL, respectively. For Y, Long and X entries the least significant word specifies the actual PMAC address to be copied.

Arguments

num_entries	Number of elements in the the_data array to be used, max 32
the_data	A pointer to an array of VBGWFormat structures which specify what is copied from DPR into PMAC's internal memory.
entry_num	Data entry to be received from table, see above for more detail

GetCurrentLabelEx

GetNumOfErrorsEx
AddErrRecordEx

COMMUNICATION APPLICATION NOTES

Common Communication Traps

Last Updated: 1/28/2003, COMTraps.DOC

1. **Timeout.** Without a doubt, timeout is what communications is all about. Increasing the timeout has solved nearly 40-50% of all communication problems encountered by end users of PMAC. What is the timeout? It is a numeric value that is proportional to the time that the host computer waits for PMAC to respond (therefore it is a host computer parameter, not PMAC). If the host doesn't wait long enough, unsychronized communications occur (causing unpredictable behavior). Applications such as the PMAC Executive Program in most cases enable the end user to modify the timeout.
2. **POBB**. The second most experienced PMAC communication trap people run into is the "PMAC Output Buffer Blues" (**POBB**). Many times programmers are unaware of PMAC's two ASCII buffers, the input and output ASCII buffer, both of which are 255 characters long ("input" and "output" are used from the perspective of PMAC, i.e. input means flow from host to PMAC and output vice versa). Regardless of which method of communication is being used the following discussion is valid.
The symptoms of POBB are that motion programs and PLCs stall, and any further incoming ASCII commands are placed into the PMAC input buffer, but do not get processed. The cause is that there are two or more pending responses for the host in PMAC's output buffer. The remedy is to purge PMAC's output buffer, with a CONTROL-X and a stripping of any extra characters waiting on the bus or serial port. Once this is done, motion programs, PLCs and any queued up commands in PMAC's input buffer are processed. Many POBB victims claim that the remedy is plugging their stalled stand-alone PMAC into a terminal (i.e. the executive program). But we know that this is just a side effect of the terminal purging PMAC's output queue.
There are many ways to get into POBB, but most are explained by the fact that people don't follow the rules –

"Do not talk to PMAC without listening"

That is, whenever a host sends PMAC an ASCII command string, the host should always check to see if PMAC has a response for the host (wait a timeout period). If I3 is set to 2 or 3 for example, PMAC will nearly always have something to say when the host sends it something. Another rule that if not followed will end up in a POBB condition is:

"Do check PMAC periodically for unsolicited responses"

If the code in PMAC has run time errors (always caused by an illegal or badly timed "CMD" statement) PMAC will send an error code string out the active ASCII port. Also if the code in PMAC has any "SEND" commands there will be ASCII responses in PMAC 's output queue. If several of these happen and the host is not checking for these unsolicited responses, POBB will occur. For those stand alone applications out there, I1 may solve this problem by allowing PMAC to purge itself. But if you have a POBB problem, it behooves you to find out what error conditions exist at run time.

3. **Initialization**. Initialization of communications is the third most often encountered problem. For those who are creating their own code, the following is a suggested procedure for initializing communication over the bus. Keep in mind that for DOS and Windows, Delta Tau does have libraries that take care of all this for you.

Serial Communications

There have been reports that people have had to use the Executive Program before their own home brewed or 3rd party serial communications software would work. The sufficient conditions for this to occur are:

1. PMAC has firmware 1.16 or before.

2. PMAC has just powered on or been reset.
3. The 3rd party software uses hardware handshaking.

From PMAC firmware version 1.16 and before, PMAC, after power up or a reset, would have it's RTS line set to FALSE, meaning it would hold off a host computer from sending PMAC commands. If the host computer would ignore the hardware handshaking for the first communication to PMAC, PMAC would then set it's RTS line to TRUE. Hardware handshaking could then be supported by the host computer. This explains why running the Executive program enabled other third party software (that uses hardware handshaking) to begin communicating. *The PMAC Executive Program does not look at the CTS line before sending PMAC commands (rather it looks at the UART's buffer empty bit instead). PMAC doesn't actively use the CTS line to hold off the host.*

Solutions: If you have firmware versions 1.16 and before and you are stuck using 3rd party software, you can easily solve this problem by shorting the host serial port's RTS and CTS lines (Pins 4 & 5 on PC's).

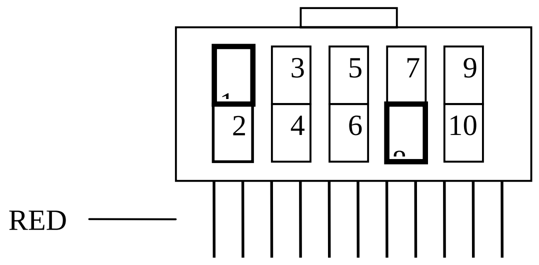
The following is a suggested initialization procedure for talking over the serial port:

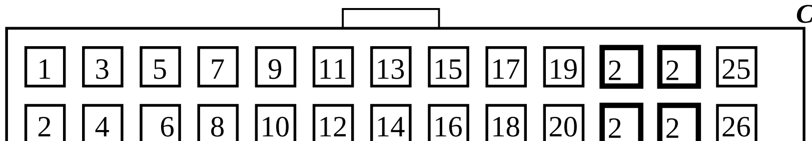
1. Send out an Control-Z and then a Control-X to activate serial communications and purge any pending ASCII communications within PMAC.
2. Send to PMAC "I1=2@1 +" to check if PMAC's have been daisy chained
3. If PMAC reports an error (i.e. "^7ERR003") because of the "+" you have confirmed that there is more than one PMAC on the serial cable.
4. If PMAC does not report an error to the "+" command then there is only one PMAC on this serial line. Send "I1 = 0" to tell PMAC to use hardware handshaking and that it is not daisychained other PMACs.

SLOW SERIAL COMMUNICATION

Subject: Correcting slow serial communication.
Applicable to: All PMAC's
Symptom: PMAC serial communication is slow.
Background:
PMAC has the ability to synchronize motors existing on different PMAC cards. When precise control is desired for motors existing on different cards, a single source for the phase and servo clock must be used. This can be accomplished by using one card as a master clock and having all other cards operate as slaves. The cards are chained together via an RS-232 cable. The phase clock and servo clock are routed to pins on the serial cable. These signals may conflict with some operating systems (WINDOWS) use of these pins for serial communications to the PC or terminal.
On PMAC 26-pin connectors, the phase clock is routed to pin 23-24 and the servo clock is routed to pin 21-22.
On PMAC2 10-pin serial connectors, the phase clock is routed to pin 1 and the servo clock is routed to pin 8.

Solution:
For serial communication to a terminal program, it is suggested that the flat serial cable be altered by cutting these wires. As shown below:





Less Common Communication Problems

1. Trying to generate an interrupt on the PC via a PMAC PLC or motion program. The problem manifests itself by getting an interrupt for every toggling of the bit, but the interrupt vector comes out to be 0. The cause here is toggling the software bit – the bit that causes PMAC to generate the interrupt – too quickly. Put a ½- or 1- millisecond delay between toggling the bits.
2. Can't communicate to a PMAC w/flash over the bus before running PMAC Executive. Solution has found to be writing a 0 to the base address of the PMAC board.
3. Serial Communications programs derived from PMAC poll don't communicate until first running the PMAC Executive Program. Remove the line which checks the CTS in the sendline() routine, i.e.:
    i. while (i++ < timeout && (inportb (combase+5) & 32) == 0);
4. Can't establish communication serially. If a Control-T character (put PMAC in full duplex) was sent to PMAC, the application you are trying to use may not be anticipating this. Put E51 (or E3 on PMAC2) and try again.

Rare Communication Problems

1. When using a PMAC2 with DPR and attempting to use interrupt based communications, setting I56 to 1 causes the DPR to become all 1s (for firmware versions 1.15G and before).
2. Control-X doesn't seem to effectively clear out PMAC's buffer. If a CTL_X is the last character in PMAC's input queue at the time PMAC checks it then all pending input and output characters in PMAC's queues get flushed. If a character gets sent after the CTL_X, making CTL_X not the last character in the queue, PMAC first parses the string. In the next background cycle PMAC will enforce the CTL_X. Because of this, even though a CTL_X was issued, if the host computer doesn't wait 100 ms or so (time depends on how busy PMAC is with higher priority tasks) PMAC could end up having characters for the host to strip.
3. Serial communications with daisy-chained PMACs. The problem is that when addressing a different card (i.e. "@2") sometimes the acknowledge character (or other requested data) is not received (if I3 is set such that an acknowledge character is expected, 2 or 3). This problem may occur because when issuing the command to switch from one card to the other, each card acts on the ASCII command in it's background cycle. PMAC processes it's background cycle tasks in whatever time is left over from higher priority tasks. If the PMAC you are addressing has very little to do in it's high priority tasks (i.e. closing loops, motion program planning, PLC0 etc) and the currently active PMAC has a lot to do in it's high priority tasks, the addressed PMAC has a high probability of becoming the active PMAC before the other PMAC became inactive. That is both PMACs become active for a short period of time. When this happens data can be lost, or perhaps an unexpected character may be received. It is therefore wise to wait a few milliseconds after addressing a card before requesting data from PMAC.
4. Serial communications with daisy chained PMACs with firmware version 1.16A and above.
   The problem is that PMAC is not responding to the command sent to it. Consider this example:.
   Let's say we've got two PMACs, PMAC_A and PMAC_B. Both have firmware version 1.16A and are daisy chained together. And to make the situation as failure prone as possible we set the baud rates on both PMAC's to 38400. PMAC_A is very busy with motion programs and PLCs, while PMAC_B is idle. Assume that PMAC_A is the currently addressed card and we wish to talk with

PMAC_B and get some motor positions while we're at it. We therefore send the following command, "@1 #1p #2p #3p #4p #5p". You may find that the above command only partially gets to PMAC_B. This is because PMAC_A will take too much time in giving up the right to receive incoming commands from the host computer, and therefore some characters are lost.

To keep from losing commands in this situation, the following procedure can be used:
1. Switch the currently addressed board to board number "x."
2. Query PMAC for the currently addressed board with the "@" command.
3. Do 2 until the appropriate board number comes up, "x" in this case.

Common Dual Ported Ram Traps

Last Updated: 1/28/2003, DPRTraps.DOC

The basic steps in setting up the DPRAM

1. For Non-Trubo PMACS configure PMAC's DPR by writing to addresses X:$786-X:$787. The Executive program should take care of this for you, but you might want to check it by hand by issuing a read command to PMAC (i.e. rhx$786,2"). For Turbo PMACs DPRAM base addresss are written in Ivariables I93 and I94.
2. Issue a save command.
3. Issue a $$$ command.
4. *PCI users do not have to worry about the DPRAM mapping as DPRAM is automatically mapped on the host computer.*

Some common Dual Ported Ram (DPR) troubles PMAC users may run into

1. The most common problem: DPR is in a location that conflicts with other devices or programs in the host computer's system. Please read the document PComm32PRO_Install.pdf for detailed instruction on how to add/remove DPRAM from the present configuration and resolve any potential conflicts with other devices.
2. Shadow Ram has not been disabled in the computer BIOS. Shadow ram has been known to cause problems with using PMAC's DPR. Disable it by configuring the BIOS settings of your computer.
3. Some computers do not report accurate memory mappings in the device manager. It is always good idea to reserve the desired memory space in the BIOS under PCI/PnP settings.
4. The contents of DPR are not what the user was expecting. PLCs may be writing to DPR or are indirectly effecting the contents/enabling of DPR. Also, check that all DPR automatic features (see DPR users manual) are disabled.
5. DPR becomes inaccessible after setting I56=1. This is a firmware glitch that has been fixed. This problem only occurred on PMAC2 for firmware versions 1.15G to 1.16A.
6. You don't have DPR. Whatever the reason, you might not have the DPR. For PMAC and PMAC-Lite the DPR is a separate board that plugs in alongside the main PMAC board. For PMAC-VME and a version of PMAC2, the DPR is located on the main board of the controller as a socketed chip. PMAC-STD users simply can't have DPR.
7. DPR Communications is flaky, noisy, or simply isn't working like it did for the last 5 years. Check and make sure the cables connecting the DPR (if your DPR uses them) are well seated. Also make sure that the daughter board is well seated on the main board. Make sure the DPR chip is well seated in it's socket if it is socketed.

Rare Problems

1. It appears as if there are two sets of ram, one the PMAC uses, the other the PC. It sounds like you haven't excluded the memory correctly. See the top of this article for setting this up.
2. DPR is fine on slow computers but not fast (P100's or faster). Or, from the host, it appears that every other byte is 0xFF. There was a modification in the programmable array logic chip for the

DPR for PMAC2's and PMAC Ultralights. Please consult factory for details on changing this chip (XYLINX chip).

Terminal Example Program

This example program is a "CONSOLE" terminal application which illustrates a creation of a simple terminal. For serial communication, checksums can be turned on and off. This sample program links with RUNTIME.CPP to accomplish runtime linking.

```
#include "runtime.h"
#include <iostream.h>
#include <string.h>
#include <conio.h>

void show_printf (char *lbuf);
void main(){
  BOOL DONE,DEVICEISOPEN=false;
  BOOL  ChecksumEnabled;
  CHAR      output_buf[256]; // holds output string
  CHAR      input_buf[16384];  // holds input string
  CHAR  out_char;
  LONG  char_counter,temp;
  DWORD dwDevice;

      if((OpenRuntimeLink()) == NULL){
            printf("Error occurred in
RuntimeLink\n");
            return;
      }
      printf("P32.EXE
\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x
1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1 Copyright DELTA
TAU 1998\n\n");
  printf("\n       Created by Allen Segall,
ALLENS@DELTATAU.COM\n");
  printf("  Use this simple terminal program for
trouble shooting your PMAC application.\n");
  printf("  NOTE: This application requires that the
PComm32PRO driver be installed.\n");
  printf("\n\n  Press CONTROL-I to initialize
communication");
  printf("\n    If you have multiple PMAC devices,
press CONTROL-I to switch between them.");
  printf("\n  Press ESCAPE to shutdown communcation
and exit.\n");
```

```
  char_counter=0;
  DONE = FALSE;
  ChecksumEnabled=FALSE;
  while(!kbhit() && !DONE)
  {
    switch(out_char = getch())
    {
      case 27: // Escape for EXIT
        DONE = true;
        break;
      case 9: // Control - I for "INITIALIZE"
        if(DEVICEISOPEN)
          DeviceClose(dwDevice);
        printf("\nWhat device number [0]?\n");
        gets(input_buf);
        dwDevice = atoi(input_buf);
        if(dwDevice <= 0)
          dwDevice = 0;
            if(!DeviceOpen(dwDevice))
        {
          printf("Device Open Failed\n");
          DEVICEISOPEN = false;
        }
        else
        {
          printf("Device Open Succeeded\n");
          printf("A terminal is active for your
use.\n");
          DEVICEISOPEN = true;
        }
        break;
      case 20: // CTRL-T Checksummed communication ON
         if(ChecksumEnabled)
         {
           printf("Checksums Disabled\n");
           ChecksumEnabled = FALSE;
         }
         else
         {
           printf("Checksums Enabled\n");
           ChecksumEnabled = TRUE;
         }

DeviceSERDoChecksums(dwDevice,ChecksumEnabled);
        break;
      case 13:
        putch(10);
        output_buf[char_counter] = '\x0';
        if(ChecksumEnabled)
        {
          temp =
DeviceSERCheckResponseA(dwDevice,input_buf,16384,outp
ut_buf);
          if(temp == 0)// An error occurred
          {
            switch(DeviceGetError(dwDevice))
            {
              case IDS_INCOMP_RECEIVE: // Unable to
get complete response
                printf("Did not receive the full
response\n");
                break;
              case IDS_CHECKSUMDATABAD:
                printf("Data checksum error\n");
                break;
              case IDS_CHECKSUMACKBAD:
                printf("Acknowledge checksum
error\n");
                break;
            }
          }
        }
        else

DeviceGetResponse(dwDevice,input_buf,16384,output_buf
);
        show_printf (input_buf);
        output_buf[0] = '\x0';
        char_counter = 0;
        break;
      default:
        output_buf[char_counter] = out_char;
        char_counter++;
        putch(out_char);
        break;
    }
  }
  if(DEVICEISOPEN)
    DeviceClose(dwDevice);
  CloseRuntimeLink();
}
void show_printf (char *lbuf)
{
      size_t            i;
      for (i = 0; i < strlen (lbuf); i++) {
            if (lbuf [i] == 13 /*||lbuf [i] == 7*/ ||
lbuf [i] == 6 ||lbuf [i] == 10)
                  switch (lbuf [i]) {
                        case 13: printf ("<CR>\r\n");
break;
                        case 7:  printf
("<BELL>\r\n"); break;
                        case 6:  printf ("<ACK>");
break;
                        case 10: printf ("<LF>\r\n");
break;
                  }
            else {
                  if (isprint (lbuf [i]))
                        putch (lbuf [i]);
                  else
                        printf ("<%u>",lbuf [i]&255);
            }
      }
}
```

INDEX

A

ASCII Communication Functions 50

B

Binary Rotary Buffer 18, 34, 37, 38, 41, 83, 84, 91, 92, 93, 37

C

Common Communication Traps............................134
Common Dual Ported Ram Traps..........................137
Configuration Functions 43
Control Panel 5, 17, 20, 21, 41, 64, 65, 66

D

Data Gathering Functions 104
Data Types, Structures, Callbacks, and Constants..48, 69–70, 122
DOWNLOADGETPROC................................59, 122
Downloading Functions...58
Downloading To PMAC........................14, 60, 61–62
DOWNLOADMSGPROC.............59–60, 59, 60, 123
DOWNLOADPROGRESS............59–60, 59, 60, 122
DPR Background Fixed Data Buffer...........27, 29, 81
DPR Background Fixed Data Buffer Initialization and Handshaking...77, 81
DPR Background Fixed Data Buffer Query Routines 78, 81
DPR Binary Rotary Motion Program Buffer...........36
DPR Numeric Read And Write 96
DPR Real Time Fixed Data Buffer Initialization ...68, 73
DPR Real Time Fixed Data Buffer Initialization and Handshaking...68
DPR Real Time Fixed Data Buffer Query Routines 68, 73
DPR Variable Background Read/Write Data Buffer 32, 83
Dpr_Bin_Rot_Errors 124
DPRTESTMSGPROC.....................................48, 123
DPRTESTPROGRESS....................................48, 124
Dual Word Conversion..96

E

Error Handling...9

F

Fixed Background Data Buffer................................18
Fixed Real Time Data buffer 17

G

Global View of the Library.......................................1
GLOBALSTATUS.................................69, 125, 126
Guide to Using ASCII Communication Functions 8

I

Initializing Communication.......................................6
Installing Pcomm32PRO 5
Interrupt Functions 108
Interrupts..13, 108

L

Linking Your Code with the Driver...........................5

M

Masking 96
Memory Organization/Correct Order of Initialization 18, 32
MOTION 23, 25, 28, 71, 75, 80, 126
MOTIONMODE 28, 30, 80, 82, 126

P

PmacAbortDPRTest 46, 48
PROGRAM ....ii, 28, 30, 80, 82, 126

R

Read/Write Functions 18

S

SERVOSTATUS..70
Shutting Down Communication 8
Standard Read/Write...96

T

Time......... 17, 18, 21, 22, 23, 24, 41, 68, 73, 125, 126

U

Usage of PComm32PRO 5
Utility Functions 118

V

Variable Functions..116